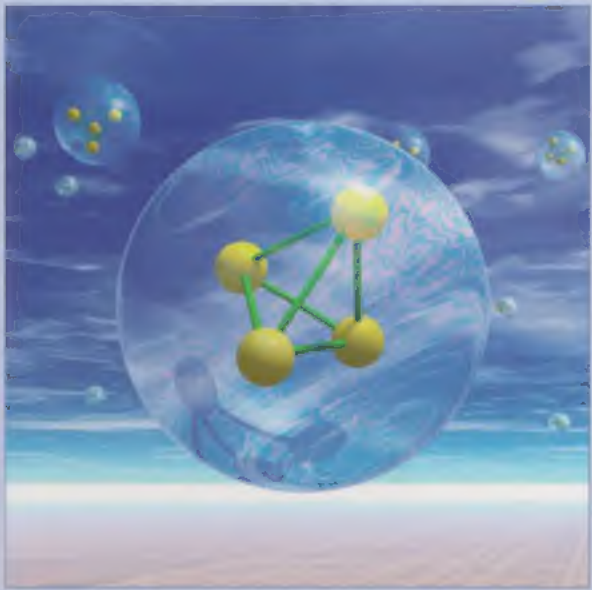
The Open University

M255 Unit 7

UNDERGRADUATE COMPUTING

Object-oriented programming with Java



Code design and class members

Unit 7

M255 Unit 7
UNDERGRADUATE COMPUTING

Object-oriented programming with Java

Code design and class members

Unit 7

This publication forms part of an Open University course M255 *Object-oriented programming with Java*. Details of this and other Open University courses can be obtained from the Student Registration and Enquiry Service, The Open University, PO Box 197, Milton Keynes, MK7 6BJ, United Kingdom: tel. +44 (0)870 333 4340, email general-enquiries@open.ac.uk

Alternatively, you may visit the Open University website at http://www.open.ac.uk where you can learn more about the wide range of courses and packs offered at all levels by The Open University.

To purchase a selection of Open University course materials visit http://www.ouw.co.uk, or contact Open University Worldwide, Michael Young Building, Walton Hall, Milton Keynes, MK7 6AA, United Kingdom for a brochure: tel. +44 (0)1908 858785; fax +44 (0)1908 858787; email ouwenq@open.ac.uk

The Open University
Walton Hall
Milton Keynes
MK7 6AA

First published 2006. Second edition 2008.

Edited and designed by The Open University.

Typeset by The Open University.

Printed and bound in the United Kingdom by The Charlesworth Group, Wakefield.

ISBN 978 0 7492 5499 5

2.1

The paper used in this publication contains pulp sourced from forests independently certified to the Forest Stewardship Council (FSC) principles and criteria. Chain of custody certification allows the pulp from these forests to be tracked to the end use (see www.fsc.org).

CONTENTS

Introduction 5

1 How work gets done in programs 6

- 1.1 Getting results by sending messages 6
- 1.2 Getting results without messages 8
- 1.3 Forms of collaboration 9
- 1.4 Overloading and overriding 13

2 Coordinating sequences of actions 16

- 2.1 Approaches to organising code 16
- 2.2 Orchestrating behaviour 18
- 2.3 The `BarnDanceCaller` class 19
- 2.4 Simulation and reality 34

3 Class methods and class variables 35

- 3.1 Class variables 35
- 3.2 Class methods 37
- 3.3 Adding a class variable and class method to the `Frog` class 40
- 3.4 Constants 44

4 A review of the different kinds of variable 46

- 4.1 Local variables 46
- 4.2 Workspace variables 46
- 4.3 Instance variables 46
- 4.4 Class variables 48
- 4.5 Method and constructor arguments 50

5 General-purpose classes 51

6 Summary 56

Glossary 58

Index 60

M255 COURSE TEAM

Affiliated to The Open University unless otherwise stated.

Rob Griffiths, Course Chair, Author and Academic Editor

Lindsey Court, Author

Marion Edwards, Author and Software Developer

Philip Gray, External Assessor, University of Glasgow

Simon Holland Author

Mike Innes, Course Manager

Robin Laney, Author

Sarah Mattingly, Academic Reader

Percy Mett, Academic Editor

Barbara Segal, Author

Rita Tingle, Author

Richard Walker, Associate Lecturer, Author and Critical Reader

Robin Walker, Critical Reader, Associate Lecturer

Julia White, Course Manager

Ian Blackham, Editor

Phillip Howe, Compositor

John O'Dwyer, Media Project Manager

Andy Seddon, Media Project Manager

Andrew Whitehead, Graphic Artist

Thanks are due to the Desktop Publishing Unit, Faculty of Mathematics and Computing.

Introduction

In this unit you will learn important new ideas about the design of object-oriented software. You will also learn more about class methods and class variables and see how they are used in practice.

The unit begins by looking at some general ideas about the design of object-oriented software. Starting with an examination of how work gets done in an object-oriented program, we then consider forms of interaction between objects, before explaining the important concepts of overloading and overriding.

Sections 2 covers ways of organising and structuring object-oriented code: We survey some standard techniques, explore trade-offs between simplicity and flexibility, and consider how the design of code may affect its maintainability. Various principles are introduced as rules of thumb for good design, and the idea of refactoring code – reorganising it to improve the design – is studied. This section of the unit ends by looking in some detail at how a special coordinating class may be added to a design in order to orchestrate the activities of a group of objects.

Section 3 revisits class methods and variables – collectively known as static members – and deals with them in depth.

The various kinds of variables available in Java are then reprised in Section 4 before going on to the final part of the unit, Section 5, where we look at the role of general-purpose utility classes that exist to provide various sorts of facilities – services – to objects of other classes, using static constants and methods.

1 How work gets done in programs

1.1 Getting results by sending messages

The principal way (though not the only way) in which useful work gets done in object-oriented programs is by objects sending messages to other objects, or indeed to themselves. Within this general pattern, there are many possible variations. Below, we will illustrate some of the main variants. The different categories that the examples illustrate are informal rather than hard and fast, and there is no need to memorise them. Many of the categories and examples overlap. However, you will find that becoming aware of the variety of ways in which sending a message can get work done will help you in understanding fundamental decisions when writing object-oriented programs.

Exercise 1

The table below illustrates some informal categories of ways of getting work done by sending a message. Fill in the blanks with message-sends (all the categories can be illustrated using frogs and/or accounts, but feel free to use any examples you like). Note that a particular message-send could be used for more than one category.

	Type of task	Example statements
1	To cause an object to take some action	`kermit.right();`
2	To get an object as a message answer	`kermit.getColour();`
3	To get a primitive value as a message answer	
4	To check whether something is `true`	
5	To change the internal state of an object	
6	To cause some object to make use of another object momentarily	
7	To cause some object to use an item of primitive data	
8	To cause the physical hardware to take some action, such as emit a sound	
9	To cause an object to become part of the internal state of another object	

Solution

Many other answers are possible – these are just examples.

These examples assume that `kermit` and `kermitJunior` are frogs and `account1` and `account2` are accounts.

	Type of task	Example statements
2	To get an object as a message answer	`kermit.getColour();` `account2.getHolder();`
3	To get a primitive value as a message answer	`kermit.getPosition();` `account2.getBalance();`
4	To check whether something is `true`	`kermit.getColour().equals(OUColour.BLUE);`
5	To change the internal state of an object	`kermit.setColour(OUColour.BLUE);` `account1.credit(100);`
6	To cause some object to make use of another object momentarily	`kermit.sameColourAs(kermitJunior);` `account1.transfer(account2, 100);`
7	To cause some object to use an item of primitive data	`kermit.setPosition(1);` `account1.setBalance(200);`
8	To cause the physical hardware to take some action, such as emit a sound	`kermit.croak();`
9	To cause an object to become part of the internal state of another object	`kermit.setColour(OUColour.PURPLE);` `account2.setholder("Wood Beez");`

Note that category 8 in the table above (causing the physical hardware to take some action) relies on the fact that a small number of methods in the Java Class Library are what is known as **native methods** which do not get work done by causing any further operations on objects or primitive data, but instead use mechanisms that cause the Java Virtual Machine in turn to cause the computer hardware take some action, such as make a sound or change the display on the screen. For example, the `croak()` method sends a message that ultimately invokes a native method. You do not need to know anything about native methods; we simply mention them so that you will be aware of their existence.

In the table above we have not considered the use of a constructor or the invocation of a class method on a class. This is because neither of these involves the sending of a message, which is what the table is looking at. Of course, the sending of a message could result in the use of a constructor or class method *indirectly*, if these appeared in the method that is executed when a message is sent. For instance, if `Frog` had a method `sayBoo()` which contained the code `OUDialog.alert("Boo!")` then the message-send `kermit.sayBoo()` would indirectly cause the invocation of the class method `alert()`.

1.2 Getting results without messages

It is also useful to consider various ways of getting work done in Java programs that do *not* involve sending a message to an object. The actions below can be performed without any message being sent.

(Of course, any of the examples below might appear inside a method, in which case they would be carried out as the result of a message being sent, and in fact this is often what happens, but the point we are making is that the actions can perfectly well be performed *without* an actual message being involved.)

As in the previous subsection, the categories are informal rather than hard and fast; they may overlap, and there is no need to memorise them. As before, we give examples for some of the categories, and ask you to come up with examples for the other ones.

Exercise 2

The table below shows some categories of ways of getting work done without sending a message. Fill in the blanks.

	Type of task	Example statement
1	Declaring a variable	
2	Invoking a constructor to create a new object	`kermit = new Frog();`
3	Assigning a value to a variable	
4	Accessing an instance variable of an object	
5	Carrying out an operation on primitive data	
6	Carrying out an operation on one or more objects without using a message	
7	Invoking a method on a class (Hint: consider the class `OUDialog`)	
8	Iterating over a block of code	
9	Selecting a branch of code to execute	

Solution

These are the examples we came up with but there are many other answers and yours are likely to be different.

	Type of task	Example statement
1	Declaring a variable	`Frog kermit;`
2	Invoking a constructor to create a new object	`kermit = new Frog();`
3	Assigning a value to a variable	`int answer = 42;`
4	Accessing an instance variable of an object	`this.position`
5	Carrying out an operation on primitive data	`2 + 2;`
6	Carrying out an operation on one or more objects without using a message	`kermit == kermitJunior;`
7	Invoking a method on a class	`OUDialog.alert("Boo");`

8	Iterating over a block of code	`int total = 0;`
		`for (int count = 1; count <= 16; count++)`
		`{`
		`    total = total + count;`
		`}`

9	Selecting a branch of code to execute	`if (count == 5) total = 42;`

Different programming languages vary in the extent to which they get things done using messages. Some languages, such as C, do not use messages at all and are not at all object oriented. Other languages, such as Smalltalk, use messages to do everything or almost everything. These are pure object oriented languages. As you can see above, Java is a hybrid that combines a mixture of object-oriented and non-object-oriented approaches. The next subsection focuses on ways of doing things in Java using objects.

1.3 Forms of collaboration

Section 4 of *Unit 2* discussed how, in order to carry out the behaviour associated with a message, an object might have to collaborate with another object to get the work done. In this subsection we look again at collaboration and identify two forms: direct interaction and indirect interaction.

One simple way in which Java objects can interact to collaborate may occur when one object is used as the argument for a message sent to some other object. For example, executing the following statement using two instances of the `Frog` class, `frog1` and `frog2`, illustrates just such an interaction.

```
frog1.sameColourAs(frog2);
```

This example is a **direct interaction** in the sense that `frog2`, which is used as an argument, is not merely ignored or just passed on by `frog1` to yet another object. Instead, within the workings of the method `sameColourAs()`, `frog1` sends a message to `frog2`, which is then used to alter the state of `frog1`. Where one object has a reference to another, and this reference is used to affect the state or behaviour of one of the objects, this is a direct interaction.

The following statement involves a more indirect interaction than the above example.

```
frog1.setPosition(frog2.getPosition());
```

In the case of this statement, `frog1` never knows of the existence of `frog2`. However, the statement has a reference to both of them and combines two messages to cause the state of one to affect, indirectly, the state of the other. In particular, the statement uses the messages `getPosition()` and `setPosition()`. The message `getPosition()` returns the position of `frog2`. The message `setPosition()` then uses the message answer as its argument, so changing the position of `frog1`. For the reasons just explained, if we were going to be precise about the type of collaboration, we would have to say that this new example is not an example of **direct interaction**, but only an example of **indirect interaction**. Some third party is coordinating the behaviour of the two frogs by sending them both messages. It is worth being aware of the distinction between direct and indirect interaction, and the fact that some kind of coordinator is needed for the latter to take place. We will return to this general idea later in the unit.

ACTIVITY 1

Open the project Unit7_Project_1.

Extend the protocol of the `Frog` class, by writing a new method with the heading

```
public void sameStateAs(Frog aFrog)
```

This method should make the state of the receiver the same as that of the argument `aFrog`. (Loosely speaking, this method will extend the behaviour of the method `sameColourAs()` that is already defined for frogs, but in this case the `position` of the receiver will also need to become the same as the `position` of the method's argument.)

Once you have written this method and got it to compile successfully, open the OUWorkspace and then an Amphibians window from the Graphical Display menu. Next execute the following code to create two frogs and two hoverfrogs.

```
Frog frog1 = new Frog();
Frog frog2 = new Frog();
HoverFrog hoverFrog1 = new HoverFrog();
HoverFrog hoverFrog2 = new HoverFrog();
frog2.setPosition(11);
frog2.setColour(OUColour.PURPLE);
hoverFrog2.setPosition(5);
hoverFrog2.setColour(OUColour.RED);
hoverFrog2.setHeight(4);
```

Next, test your new method by executing the following four statements (one by one).

```
frog1.sameStateAs(frog2);
hoverFrog1.sameStateAs(hoverFrog2);
frog2.sameStateAs(hoverFrog1);
hoverFrog2.sameStateAs(frog1);
```

View the results of these statements in the Amphibians window and by inspecting the receivers of the `sameStateAs()` messages after each statement is executed.

The code for the `sameStateAs()` *method has been added to the* `Frog` *class in Unit7_Project_2. So, if you had problems getting your* `sameStateAs()` *method to compile, the above code can be executed in the OUWorkspace after opening Unit7_Project_2.*

DISCUSSION OF ACTIVITY 1

There are several ways of writing the method, but here is one possibility (we have omitted the method comment for brevity).

```
public void sameStateAs(Frog aFrog)
{
    this.sameColourAs(aFrog);
    this.setPosition(aFrog.getPosition());
}
```

Notice how we have made use of the `sameColourAs()` message to set the `colour` of the receiver – this is much better style than using `this.setColour(aFrog.getColour());`

In the case of the frogs `frog1` and `frog2`, the effect of the message-send

```
frog1.sameStateAs(frog2);
```

is to put the receiver into the same state as the argument.

In the case of the hoverfrogs `hoverFrog1` and `hoverFrog2`, the effect of the message-send

```
hoverFrog1.sameStateAs(hoverFrog2);
```

is to put the receiver into the same state as the argument as regards its `position` and `colour` instance variables, but leaves `height` untouched

The message-send

```
frog2.sameStateAs(hoverFrog1);
```

sets the `postion` and `colour` instance variables of `frog2` to be the same as `hoverFrog1`, and finally

```
hoverFrog2.sameStateAs(frog1);
```

sets the `postion` and `colour` instance variables of `hoverFrog2` to be the same as `frog1` leaving the height of `hoverFrog2` untouched

As the previous activity has demonstrated, the message `sameStateAs()`, as defined by the corresponding method in the `Frog` class, can be used to put two frogs in the same state. The previous activity also demonstrated, which you may have found surprising, that this message can also be sent to a `HoverFrog` object with another `HoverFrog` object as the argument, or to a `Frog` object with a `HoverFrog` object as the argument, or to a `HoverFrog` object with a `Frog` as the argument

This flexibility is possible because `HoverFrog` is a subclass of the `Frog` class and so the `HoverFrog` class inherits all of the methods defined for the `Frog` class, therefore the message `sameStateAs()` can also be sent to `HoverFrog` objects Also although the formal argument to `sameStateAs()` has been defined to be of type `Frog`, we can use a `HoverFrog` object as the actual argument because `HoverFrog` is a subtype of `Frog`. Hence, hoverfrogs are substitutable for frogs in both places – both as the receiver of the message and as the argument.

Note the terminology we are using, variables and formal arguments have a type, whereas the objects that are assigned to them have a class

However frogs have only two instance variables, `position` and `colour` whereas hoverfrogs have an extra instance variable `height`, which the method `sameStateAs()` knows nothing about. Therefore a message-send such as

```
hoverFrog1.sameStateAs(hoverFrog2);
```

cannot result in the two hoverfrogs having the same state unless they had the same value for `height` in the first place Only the values of the `position` and `colour` instance

variables of `hoverFrog1` would be changed to match those of `hoverFrog2`, `height` would be ignored and remain unchanged as the `sameStateAs()` method knows nothing about the `height` instance variable.

A message-send such as

```
frog2.sameStateAs(hoverFrog1);
```

can be used to put `frog2` in the same state as `hoverFrog1` as regards the kinds of state that frogs know about – `position` and `colour`. However, frogs do not have a `height` instance variable, so `frog2` cannot be put at the same height as `hoverFrog1`.

Finally

```
hoverFrog2.sameStateAs(frog1);
```

In this case, `hoverFrog2` can be put in the same state as `frog1` as regards `position` and `colour`, but `hoverFrog2`'s `height` instance variable will be left untouched, since the object referenced by `frog1` does not have a `height`, and more to the point, the method `sameStateAs()` does not know about `height`.

Strictly speaking, there is no possibility whatsoever for a hoverfrog and a frog to have the same state as they have a different number of instance variables. However it is possible to write a method that will put two hoverfrogs into the same state. To do this, we need to define a method `sameStateAs()` in the `HoverFrog` class, which will make the `height` instance variable of the two hoverfrogs the same as well as their `position` and `colour`.

ACTIVITY 2

Open the project Unit7_Project_2.

Extend the protocol of the `HoverFrog` class, by writing a new method with the heading

```
public void sameStateAs(HoverFrog aHoverFrog)
```

which takes `height` into consideration (you may find it helpful to refer back to Activity 1).

Once you have written this method and got it to compile successfully, open the OUWorkspace, and then an Amphibians window from the Graphical Display menu. Next execute the following code to create two hoverfrogs with different states.

```
HoverFrog hoverFrog1 = new HoverFrog();
HoverFrog hoverFrog2 = new HoverFrog();
hoverFrog2.setPosition(5);
hoverFrog2.setHeight(3);
hoverFrog2.setColour(OUColour.RED);
```

Next test your new method by executing the following statement:

```
hoverFrog1.sameStateAs(hoverFrog2);
```

View the result of executing the statement in the Amphibians window and by inspecting `hoverFrog1`.

The code for the `sameStateAs()` method has been added to the `HoverFrog` class in Unit7_Project_3. So, if you had problems getting your `sameStateAs()` method to compile, the above code can be executed in the OUWorkspace after opening Unit7_Project_3.

DISCUSSION OF ACTIVITY 2

There are several ways of writing the method but here is one possibility (we have omitted the method comment for brevity)

```
public void sameStateAs(HoverFrog aHoverFrog)

   // First use the superclass method to make the position and
   // colour of the two hoverFrogs the same
   super.sameStateAs(aHoverFrog);
   // Then make their heights the same
   this.setHeight(aHoverFrog.getHeight());
```

Here we are making use of the fact that the method with the signature `sameStateAs(Frog)` defined in the `Frog` class already allows the `position` and `colour` of `HoverFrog` objects to be set to those of the given argument. To invoke that method in the superclass (`Frog`) at run-time we simply send the `sameStateAs()` message to the receiver via the pseudo-variable `super`. Then, to complete our new `sameStateAs()` method, we merely need to set the receiver's `height` to be the same as the argument's `height`. Note the element of code reuse in this example

You could make the method work by reproducing the code from `sameStateAs()` in `Frog` like this

```
public void sameStateAs(HoverFrog aHoverFrog)

   this.sameColourAs(aHoverFrog);
   this.setPosition(aHoverFrog.getPosition());
   this.setHeight(aHoverFrog.getHeight( );
```

but that would be poor design and should always be avoided for reasons which will be discussed in detail in Section 2

1.4 Overloading and overriding

In *Unit 6* we looked at how inherited methods could be overridden in a subclass We also looked at how constructors could be overloaded In this subsection we look at how *methods* are overloaded and investigate how such methods are selected for execution at run-time, contrasting this with how overridden methods are selected for execution

Notice in the previous activity, that the `HoverFrog` class now has *two* methods defined whose method name is `sameStateAs()`, and both of them have a single argument These two methods have different signatures, as the *types* of their argument differ – in one case the argument is of type `Frog`, in the other case the argument is of type `HoverFrog` The latter method is defined directly in `HoverFrog`, whereas the former method is inherited from `Frog`

In any case where a class has more than one method with the same name, but the methods differ in the *number* of arguments, or in the *order* or *type* of one or more arguments, the method name is said to be **overloaded** In the present example the number of arguments is the same – one – but as noted the type is different.

ACTIVITY 3

Open the project Unit7_Project_3.

Execute the following statements in the OUWorkspace to create two frogs and two hoverfrogs all with different states.

```
Frog frog1 = new Frog();
Frog frog2 = new Frog();
HoverFrog hoverFrog1 = new HoverFrog();
HoverFrog hoverFrog2 = new HoverFrog();
frog2.setPosition(11);
frog2.setColour(OUColour.PURPLE);
hoverFrog2.setPosition(5);
hoverFrog2.setColour(OUColour.RED);
hoverFrog2.setHeight(4);
```

Next execute the following statements (one by one):

```
frog1.sameStateAs(frog2);
hoverFrog1.sameStateAs(hoverFrog2);
frog2.sameStateAs(hoverFrog1);
hoverFrog2.sameStateAs(frog1);
```

View the results of these statements in the Amphibians window and by inspecting the receivers of the `sameStateAs()` messages after each statement is executed.

DISCUSSION OF ACTIVITY 3

The effect of the message-send

```
frog1.sameStateAs(frog2);
```

is to invoke the method `sameStateAs()` defined in the `Frog` class and to put the receiver (a frog) into the same state as the argument (here also a frog).

In the case of the message-send

```
hoverFrog1.sameStateAs(hoverFrog2);
```

the compiler realises that the `HoverFrog` class has two candidate methods that could match the message-send. The method with the signature `sameStateAs(Frog)` is inherited from the `Frog` class, and the method `sameStateAs(HoverFrog)` is defined by the `HoverFrog` class itself. Either method would match the message because `HoverFrog` is a subtype of `Frog`, so it is legal for a `HoverFrog` object to be supplied as the argument to a method wherever a `Frog` instance is expected.

In a case like this, when a class has several overloaded methods, then the compiler must decide which method signature the JVM should use to select a method at run-time. The process of picking the best match from a set of candidate methods is called **overload-resolution**. The compiler chooses the best signature match based on the compile-time *type* of the actual argument provided to the message. In the case of the message-send

```
hoverFrog1.sameStateAs(hoverFrog2);
```

`hoverFrog2`, which is used as the argument to the `sameStateAs()` message, has been declared as a variable of *type* `HoverFrog`. Hence the compiler produces code that instructs the JVM to select the method that matches the signature `sameStateAs(HoverFrog)` at run-time. At run-time the method `sameStateAs()` defined in the `HoverFrog` class is invoked which puts the receiver (a hoverfrog) into the same state as the argument (here also a hoverfrog), taking into account their respective `height`.

The message-send

```
frog2.sameStateAs(hoverFrog1);
```

is non-problematic, the `Frog` class has only one `sameStateAs()` method, so there is no confusion. `HoverFrog` is a subtype of `Frog`, so it is legal for a `HoverFrog` object to be supplied as the argument wherever a `Frog` instance is expected. The version of the method that is invoked is the one from `Frog`, because at run-time that is the class of the receiver. The `position` and `colour` of the receiver are set to those of the argument, but, of course, no attempt is made to set a `height`, which could not apply to `frog2` since it references a `Frog` object.

In the case of the message-send

```
hoverFrog2.sameStateAs(frog1);
```

the compiler again realises that the `HoverFrog` class has two candidate methods that could match the message-send. So once more overload-resolution takes place and as the reference provided as the argument to the message (`frog1`) has been declared as a variable of *type* `Frog` the compiler produces code that instructs the JVM to select the method that matches the signature `sameStateAs(Frog)` at run-time. So we infer that the method that is invoked will be the version inherited from `Frog`. Hence only the `position` and `colour` instance variables of the receiver are set to those of the message's argument and no attempt is made to set the `height`.

It is important at this juncture to make clear the difference between how an overloaded method and an overriden method are invoked at run-time. A method name is said to be **overloaded** if another method that can be invoked by a message to the same type of receiver has the same name, but the arguments differ in number, order or type. By contrast, if a method has the same name and the same arguments (and return type) as an accessible method in a superclass, it is said to **override** that method.

Whenever a message corresponding to [illegible] or more overloaded methods is detected at compile-time, the overload-resolution [illegible] the compiler decides what method signature the JVM should [illegible] at run-time, based on the compile-time types of the actual arguments provided in the message. The run-time *class* of the arguments cannot later affect which of the overloaded methods is invoked at run-time – it is fixed at compile-time.

However, in the case of *overriding*, while the declared compile-time *type* of the variable referencing a receiver is known to the compiler, the run-time *class* of the receiver is generally not, and it may make a vital difference because different classes of receiver will each implement their own different method corresponding to the message in question. Hence the appropriate choice of method will depend on the *class* of the message receiver at run-time.

Thus, there is a stark and important difference between the ways in which *overriding* and *overloading* affect the selection of methods to correspond to a given message. This contrast may also be seen, if you prefer, as being between the ways in which the type of *message arguments*, as opposed to the class of a *message receiver*, affect which method will be invoked by a given message. In either case, the *class* of the arguments is never considered for the purpose of choosing a method – the number, type and order of the arguments is all that Java is designed to consider.

Of course, it is possible for a message to be affected by both overloading and overriding, in which case part of the decision on which method signature to choose will be taken at compile-time, but a final decision will not in general be taken until run-time.

2 Coordinating sequences of actions

The examples explored in the previous section illustrate that there are diverse ways, in an object-oriented program, in which work can get done. However, in any program, if it is to be understandable and maintainable, actions (statements) need to be structured into well-organised wholes. So, as well as an awareness of the different ways in which things can get done, we need to consider how sequences of actions can be organised and tied together.

In the next subsection we will reprise three ways of organising a sequence of actions which you have already encountered. We will then go on to introduce you to a fourth, new way of organising a sequence of actions.

2.1 Approaches to organising code

Organise a complex sequence of actions as a single method

Perhaps the simplest way to unify or organise a complex sequence of actions or statements is to *write it as a method of the appropriate class*. We have already used this approach on many occasions.

The dancing frogs of *Unit 5* provide a good opportunity to illustrate the idea. Getting a frog into its starting position in the traditional manner for a dance involves a sequence of several actions, which in *Unit 5* were merely assembled in the OUWorkspace. Assembling these actions into a single method `takeUpYourPositions()`, as you will be asked to do shortly, is a neat way of unifying the sequence.

Although this is a quite straightforward approach, it is highly effective.

Organise a complex sequence of actions as two or more methods

The second approach is simply a variant of the first. The hallmark of this variant is where a sequence of statements *could* be written as a single method of a class, but is better split into *several methods of the same class*. There are several reasons why this might be desirable. If the method is long, breaking it up might be desirable simply to improve clarity and readability. But a more compelling reason is typically to remove code that would otherwise be duplicated.

For example, any number of frog dances start with the movements necessary to get into position. But it would be very poor programming style, indeed, to encode such dances by *copying and pasting* code from the `takeUpYourPositions()` method mentioned above into each of those dance methods. Instead, of course, the more complicated dance methods should simply include the message

```
this.takeUpYourPositions();
```

which will have the same effect, but in a much neater way. Then, if any future need arises to change the movements necessary to get into position, this need only be done in one single place (namely the `takeUpYourPositions()` method), instead of having to track down multiple places where the contents of the method might have been copied and pasted.

Indeed, whenever two or more methods have some section of duplicated code in common, it is usually a good idea to factor out the duplicated code as a separate method, and have the original methods call this code by sending the corresponding message to `this`

Recall from *Unit 6* that methods which have no other purpose than to be used by other methods in the same class, are known as **helper methods** Helper methods are normally not part of the external protocol of the class so are usually declared as `private`. However, sometimes methods created in this way are made public (if they can be useful to other objects as well), in which case they would not be regarded as helper methods

For our present purposes the above discussion simply illustrates that when packaging up a complex sequence of actions as a method it is always worth considering whether clarity can be improved or duplicated code avoided by breaking up the method into two or more methods. More generally, this is a special case of the universal design principle that duplicated code should be eliminated whenever possible. This is a fundamental principle for writing good quality code

Design principle 1
Write only once - duplicated code should be eliminated whenever possible

Distribute complex actions over appropriate classes

In the case of a single dancing frog only a single class is involved so it is fairly obvious that the right way to organise a complex sequence of actions is to write it either as a single method of the `Frog` class or to split it over several methods of the class, if this improves clarity or avoids duplication

However sometimes a complex sequence of actions systematically involves objects of *more than one class*. (For example, the sequence of actions involving instances of `WeatherFrog`, `Daisy` and `MetOffice` in *Unit 6*.) In general, the recommended approach in such cases is to *avoid detailed centralised control of the sequence of actions* in a single central 'director' or 'control' method that tells each of the objects of different classes what to do in detail In general, it is far better for each class of object to retain as much autonomy as possible Any object that must tell another object what to do should wherever possible restrict itself to communicating the general idea and let the receiving object work out the details for itself

For example, in the example illustrated by the classes `MetOffice`, `WeatherFrog` and `Daisy`, as explored in *Unit 6*, a change in the weather could, in principle, trigger an arbitrarily complicated sequence of actions. One could imagine a misguided redesign of this example whereby an instance of `MetOffice` sent detailed messages to `WeatherFrog` and `Daisy` objects telling them step-by-step how to respond to each change in the weather

In fact, however, the `MetOffice` class is wisely designed neither to know nor to care anything about `Daisy` and `WeatherFrog` objects and the actions that they may perform in response to changes in the weather An instance of `MetOffice` simply assumes that its clients can respond to the messages `rain()` and `sun()` and leaves it to the clients to deal with the detail of how to respond

This is a good example of organising a potentially complex sequence of actions by distributing responsibility over the relevant classes The ideal is for the knowledge of a detailed sequence of actions involving objects of several classes to be *distributed* so that each object knows how to play its own part, and does not try to interfere with the detail of how other objects go about their business This is an important principle of design, though it is difficult to state precisely in general terms Application of this principle can be much more of a matter of opinion than, for example, detecting duplicated code However the principle is equally useful You will see a worked example of it in action later in the unit

Design principle 2
Distribute responsibility - objects should have autonomy to deal with matters that concern them. Objects should *avoid interfering*, as far as possible, with the *fine detail* of how other objects go about their business

This principle of distributing responsibility is the normal way to organise object-oriented programs and is fundamental to object-oriented development

Coordinate actions by identifying and using a missing class

The fourth and final approach to organising a complicated sequence of actions may seem at first to contradict the third approach of distributing responsibility among existing classes, as it involves creating a new class of object specifically to handle the organisation of a sequence of actions. However there is not really a contradiction.

For example, if you wanted to organise a frog dance that involved *more than one* frog carrying out coordinated actions, then why should any *single* frog be doing all of the organising. Which frog would one choose? Perhaps a new class of objects should be created to do the coordinating – call it the `BarnDanceCaller` class (we will ask you to explore this idea practically in the activities below)

It would be poor design if such a `BarnDanceCaller` class had an instance method that spelled out every single action by every frog, but it would be perfectly reasonable if each frog knew how to do various dance sequences and the caller simply coordinated them by saying which sequences to do in turn. There is no clear need for a dancer to know anything about any object other than itself

So if there is a group of objects to be coordinated it is often a good approach to create a new class of object to do the job as long as the responsibility remains as distributed as possible, and the central object abstains from unnecessary micro-managing of the objects being coordinated.

The key point is that sometimes an organising or orchestrating class is needed to tie together the different parts of a complicated interaction that does not seem to belong to any single one of the objects obviously involved. If you tried to ignore the missing class and tried to organise the actions by distributing code among the existing objects some of the objects would end up having to deal with things that did not seem to be any of their business

You do not need to memorise the four different approaches described above and we do not expect you to be able to decide infallibly which one is called for. But you should be able to discuss informally the pros and cons of the different options in particular cases. Which approach will work best is rarely cut and dried and different solutions may be equally valid – the important thing is to be aware of the possibilities.

SAQ 1

In the discussion of Activity 2, you were told that it is very much better style to reuse code from the superclass, by sending a message to `super`, than it is to duplicate the code from the superclass. Which of the ideas discussed above is this an example of?

ANSWER..

Various answers are possible but the most obvious is Design principle 1. Using `super` instead of repeating the statements from the superclass method is a good example of avoiding code duplication.

2.2 Orchestrating behaviour

As discussed above, sometimes it is necessary to provide an object that makes sure some collection of objects carries out one or more complicated sequences of actions in the right order. In some cases this is not required, and nothing more is needed than a

single method in an existing class, or a set of methods distributed over one or more classes. Generally it is better to distribute responsibility if possible while making sure duplication of code is minimised or eliminated. However, as indicated, sometimes an organising object is useful.

In cases where an organising or coordinating object is used this object is known as an orchestrating object or **orchestrating instance** which is often the sole instance of a class which models the way in which other objects can be coordinated. An orchestrating instance coordinates the interactions between objects under its supervision by sending the appropriate messages to them. To fulfil its role, an orchestrating instance must encode one or more sequences of interactions between participating objects in one or more methods.

In order to communicate with the objects under its supervision via messages, an orchestrating instance needs an instance variable to reference each of the objects involved.

Strictly speaking one cannot send a message to an instance variable and we really ought to say 'send a message to an object via a reference held by an instance variable'. However, this is a bit wordy, so for convenience we speak of sending a message to a variable, as an informal shorthand.

Thus an orchestrating object can communicate with a particular object by sending a message to the instance variable that references that object. For example, if some orchestrating instance has an instance variable frog1 which references a frog, we can use statements such as the following

```
frog1.right();
```

to make that particular 'constituent object' move right.

In order to synchronise the interactions between constituent objects an orchestrating instance will hold details of a particular sequence of interactions as methods. The task concerned can be initiated simply by sending the appropriate message to the orchestrating instance.

In the following subsection you will learn more about object orchestration by exploring barn dancing in the Amphibian world. You will coordinate various amphibians moving in a set of simple dances across the stones in their pond. This will build on previous work in *Unit 5* to develop an orchestrating `BarnDanceCaller` class.

2.3 The `BarnDanceCaller` class

In *Unit 5* you saw how to use `for` and `while` loops to make frogs perform a simple set of dances. We will see that such activities can be organised in a neater and more manageable fashion by using an orchestrating `BarnDanceCaller` class, an instance of which will represent the coordinating role of the caller in a barn dance.

When you have completed the practical activities in this subsection you should be able to send simple messages to the orchestrating instance of `BarnDanceCaller` to request that a specific dance routine be performed. The orchestrating instance will then pass on more detailed messages requesting the necessary movements from the frogs participating in that dance. We will use this to illustrate in more detail the notions of collaborating and orchestrating objects, and state-dependent behaviour.

As you know, when defining a new class, you generally need to consider the following points.

- What its superclass will be.
- Any additional instance variables required.
- What sort of constructor is needed.
- Any additional methods (behaviour) required and any inherited methods that needed to be overridden.

SAQ 2

What should be the superclass of `BarnDanceCaller`?

ANSWER

`BarnDanceCaller` should be a subclass of `Object` since there does not seem to be an existing class from which you would want to inherit some behaviour (methods) and state (instance variables).

If you proposed making `BarnDanceCaller` a subclass of `Frog` or `Amphibian` you probably reasoned that an instance of this class should represent a special type of `Frog` that has additional behaviour so that it could act as a caller in a barn dance.

This is a perfectly understandable approach, because it is natural to think 'Won't I need `BarnDanceCaller` to inherit from `Frog` so that I will have access to `left()` and `right()` and so forth?'

But in fact there is no need – the `BarnDanceCaller` class itself does not need to be able to move left and right – it just needs to have references to the two frogs so that it can tell *them* to do that. Its sole purpose is to organise other objects and it does not need to be able to do the things they do.

SAQ 3

What instance variables should `BarnDanceCaller` have?

ANSWER

An orchestrating instance needs to have a means of locating all the objects under its supervision so that it can communicate with them in order to coordinate the interactions between them. Since an instance of `BarnDanceCaller` is required to coordinate two frogs to move in a set of simple dances, you will need two instance variables to hold references to the dancers, say, [illegible] and [illegible]. Sending messages to the objects referenced by these instance variables will make the dancing frogs move as required.

A `BarnDanceCaller` does not itself need to have a `colour` or a `position` – it is sufficient that the `BarnDanceCaller` knows about frogs, which do have these attributes.

Having tried these SAQs, you should now start Activity 4.

ACTIVITY 4

Open the project Unit7_Project_4.

Create a new class called `BarnDanceCaller` and comment it. Pay particular attention to the provision of instance variables to allow the barn dance caller to refer to the two dancers. You should write setters and getters for both instance variables, but *those are all the instance methods you are asked to write for the moment*. We will add other behaviour in the activities which follow.

DISCUSSION OF ACTIVITY 4

Here is our code for the class so far (comments omitted for brevity)

```
public class BarnDanceCaller

   // instance variables
   private Frog dancer1;
   private Frog dancer2;

   public BarnDanceCaller()

      super();


   public void setDancer1(Frog aFrog)

      this.dancer1 = aFrog;


   public Frog getDancer1()

      return this.dancer1;
   }

   public void setDancer2(Frog aFrog)
   {
      this.dancer2 = aFrog;


   public Frog getDancer2()

      return this.dancer2;
```

In Activity 4, you created the orchestrating BarnDanceCaller class for the barn dance with two instance variables dancer1 and dancer2, so that an instance of BarnDanceCaller can hold a reference to each of the dancers participating in a dance As we want to give any two frogs the chance to dance together we need to provide the BarnDanceCaller class with a method to instruct the orchestrating instance which frogs will be dancing – we shall do this in the next activity.

ACTIVITY 5

Open Unit7 Project 5 This project includes the BarnDanceCaller class as developed so far in Activity 4.

Write an instance method of BarnDanceCaller with the following method heading

```
public void setDancers(Frog frog1, Frog frog2)
```

This method should make use of the setter methods you wrote in the previous activity to set the instance variables dancer1 and dancer2 to the arguments of the setDancers() method

Once you have written the method and successfully re-compiled the BarnDanceCaller class, open the OUWorkspace and execute the following statement

```
BarnDanceCaller caller = new BarnDanceCaller();
```

Inspect caller to confirm that the instance variables dancer1 and dancer2 do not reference Frog objects but hold the value null.

Now create two frogs using the following code:

```
Frog sam = new Frog();
sam.setColour(OUColour.PURPLE);
Frog lew = new Frog();
lew.setColour(OUColour.YELLOW);
```

Now execute the following statement to set the instance variables dancer1 and dancer2 to the frogs referenced by sam and lew.

```
caller.setDancers(sam, lew);
```

Finally, inspect caller to confirm that the instance variables dancer1 and dancer2 now each reference a Frog object.

The code for the setDancers() method has been added to the BarnDanceCaller class in Unit7_Project_6. So, if you had problems getting your setDancers() method to compile, the above code can be executed in the OUWorkspace after opening Unit7_Project_6.

DISCUSSION OF ACTIVITY 5

The following is one way of writing the method setDancers() for BarnDanceCaller:

```
public void setDancers(Frog frog1, Frog frog2)
{
   this.setDancer1(frog1);
   this.setDancer2(frog2);
}
```

This is a straightforward method to establish which dancers (that is, which frogs) an instance of BarnDanceCaller will be orchestrating.

It is useful to summarise all of the statements in this activity needed in testing the method, with comments, which will remind you later of what you did. This might be as follows

```
// Create a caller
BarnDanceCaller caller = new BarnDanceCaller();
// Create two dancers and modify their state
Frog sam = new Frog();
sam.setColour(OUColour.PURPLE);
Frog lew = new Frog();
lew.setColour(OUColour.YELLOW);
// Give the caller new dancers
caller.setDancers(sam, lew);
```

After executing this statement series, inspection of caller should confirm that its instance variables refer to two frogs as anticipated

In *Unit 5*, you developed a series of statements which, when executed, would cause a frog to hop from any stone to the central stone (position 6), and then turn red. Since this needs to be performed prior to each dance it would be convenient to wrap up the statement series as a method. In fact, in Exercise 9 in *Unit 5*, this was done using a method of the `Frog` class. For the purposes of this section, we will ignore the method of *Unit 5*.

Exercise 3

Imagine that a colleague proposes to write a method of `BarnDanceCaller`, called `takeUpYourPositions()`, which will cause both frogs to hop to the central stone and turn red. Your colleague suggests the following code.

```
/**
 * Causes both dancers to take up their positions
 */
public void takeUpYourPositions()
{
   // dancer1 hop to centre and turn red
   while(this.getDancer1().getPosition() < 6)
   {
      this.getDancer1().jump();
      this.getDancer1().right();
   }
   while (this.getDancer1().getPosition() > 6)
   {
      this.getDancer1().jump();
      this.getDancer1().left();
   }
   this.getDancer1().setColour(OUColour.RED);

   // dancer2 hop to centre and turn red
   while(this.getDancer2().getPosition() < 6)
   {
      this.getDancer2().jump();
      this.getDancer2().right();
   }
   while (this.getDancer2().getPosition() > 6)
   {
      this.getDancer2().jump();
      this.getDancer2().left();
   }
   this.getDancer2().setColour(OUColour.RED);
}
```

Assuming that this code makes the frogs move in the right way, would this be a good way to organise this code, focusing on the approaches and principles noted in Subsection 2.1. Which principles from that subsection seem to apply to this code?

Solution

Although the above method meets the requirements in terms of frog movements, it is in very poor style, according to the criteria of Subsection 2.1 The most obvious stylistic shortcoming is that the code of the main statement series – the code to move the frog dancer – is *duplicated* An important part of good style, in general is to avoid code duplication, for reasons discussed below You may have noticed that there is at least one *other* major stylistic fault in the method as written above based on the criteria of Subsection 2 1, but for now we will focus exclusively on the code duplication

As a consequence of the proposed code duplication, if the programmer subsequently decided that, for example, the dancers should commence from the home stone and perform the dance while yellow, then it would be necessary to make the required modifications in two places. The presence of duplicated code increases the effort to maintain the method and increases the possibility of errors being introduced when the method is subsequently modified. In this way, removing code duplication makes code more flexible and easier to maintain Ease of maintenance (or its absence) is such an important property of code that programmers use a specific term, **maintainability** to refer to this property

ACTIVITY 6

Having put the above arguments to your colleague they ask you to modify the code to remove the code duplication, before making any other stylistic changes.

Open the project Unit7_Project_6.

This contains the code for the [illegible] class as specified in Activities 4 and 5 and the badly designed version of the method `takeUpYourPositions()` as given in Exercise 3.

Open the OUWorkspace and the Graphical Display.

As in Activity 5, in the OUWorkspace create an instance of `BarnDanceCaller`. Next create two `Frog` objects and then send a `setDancers()` message to your `BarnDanceCaller` object to set its instance variables to the two frogs.

Then send your `BarnDanceCaller` object the message `takeUpYourPositions()` to test that it does in fact produce the right frog movements.

Now rewrite the code of the [illegible] method so that the duplicated part becomes a separate helper method called [illegible] which is used where required within [illegible] Remember that helper methods should be declared as [illegible]

Test your rewritten method in the OUWorkspace to confirm that it still orchestrates the correct frog movements.

DISCUSSION OF ACTIVITY 6

Our solution is shown here

```
/**
 * Causes both dancers to take up their positions.
 */
public void takeUpYourPositions()
{
   this.getReady(this.getDancer1());
   this.getReady(this.getDancer2());
```

```
/**
 * Causes a frog to hop to its central stone and turn red.
 */
private void getReady(Frog aFrog)
{
   while(aFrog.getPosition() < 6)
   {
      aFrog.jump(); aFrog.right();
   }
   while (aFrog.getPosition() > 6)
   {
      aFrog.jump(); aFrog.left();
   }
   aFrog.setColour(OUColour.RED);
}
```

In Exercise 3 you began with working code that met its requirements. In Activity 6 you rewrote the code in such a way that it ended up having exactly the same effect as before, but code duplication was removed, style was improved, and the class became easier to maintain. When code is rewritten without changing its overall effect, but for the purpose of improving its design, removing code duplication, or improving maintainability, the process is called **refactoring**. You encountered the term refactoring in *Unit 6* when the term was used in the context of changing the hierarchy of the amphibian classes by introducing the abstract class `Amphibian` as the common superclass for the `Frog`, `Toad` and `HoverFrog` classes.

Exercise 4

The refactored code seen in the discussion of Activity 6 is a great improvement on the code proposed in Exercise 3. But it still violates the design principles noted in Subsection 2.1. What further refactoring could be done to improve the design?

Solution

The code organisation of both Exercise 3 and Activity 6 violates the design principle noted in Design principle 2 given in Subsection 2.1. The [illegible] is interfering too much with the fine detail of how frogs go about their business. We have made `getReady()` a method in `BarnDanceCaller`, which means the latter dictates to the frogs what steps they must perform, but frogs should decide for themselves the fine detail of how they take up their positions. As we have already noted, the application of this principle can be much more of a matter of opinion than, for example, detecting and removing duplicated code. People can reasonably differ on how to interpret the second principle. However, its use is still valuable.

You may have found other ways in which the design of the code is poor and needs refactoring. However, for the moment we will focus on applying the second design principle.

ACTIVITY 7

Open the project Unit7_Project_7.

This contains the code for the `BarnDanceCaller` class as it was at the end of Activity 6.

Relocate the method `getReady(Frog aFrog)` to the `Frog` class. It will no longer need an argument, because a frog 'knows' who it is without being told. You simply need to replace `aFrog` in the method code with `this`.

Rewrite the `BarnDanceCaller` method `takeUpYourPositions()` so that it now sends `getReady()` messages to the dancers telling them to get themselves ready. Note that as an instance of `BarnDanceCaller` now needs to send a `getReady()` message to a `Frog` object, this time the `getReady()` method should be declared as `public`.

Test your refactored code in the OUWorkspace.

DISCUSSION OF ACTIVITY 7

Here is our refactored code.

In the `BarnDanceCaller` class

```
/**
 * Causes dancer1 and dancer2 to take up their positions.
 */
public void takeUpYourPositions()
{
   this.getDancer1().getReady();
   this.getDancer2().getReady();
}
```

In the `Frog` class

```
/**
 * Causes the receiver to hop to its central stone and turn red.
 */
public void getReady()
{
   while (this.getPosition() < 6)
   {
      this.jump(); this.right();
   }
   while (this.getPosition() > 6)
   {
      this.jump(); this.left();
   }
   this.setColour(OUColour.RED);
}
```

The code for this activity has been added to the `Frog` *class and the* `BarnDanceCaller` *class in Unit7_Project_7_sol.*

SAQ 4

Notice that the code above uses messages to the pseudo variable `this` a lot, and uses getter messages such as `getDancer1()` to access instance variables such as `dancer1`. Hence our code uses statements such as

```
this.getDancer1().getReady();
```

whereas in some common Java programming styles it would be more common to see shorter statements such as

```
dancer1.getReady();
```

What are advantages of the two approaches?

ANSWER

The shorter form is quicker to write, more concise, and can be easier to read. However, use of getters and setters gives much improved flexibility as the place and manner in which state is stored can be subsequently changed without having to change code that uses it. In addition the use of the pseudo variable `this` (which is not strictly necessary) explicitly makes it clear that the receiver is sending itself a message, and helps make the code easy to understand.

Exercise 5

Refactoring to remove code duplication is the simplest way to improve the design of your code. However, there are ways of misunderstanding and misapplying the idea of refactoring. Consider the following method.

```
public void getReady()

    while(this.getPosition() < 6)

        this.jump(); this.right();
    }
    while (this.getPosition() > 6)

        this.jump(); this.left();
    }
    this.setColour(OUColour.RED);
```

A colleague asks you: why do we not use `getPosition()` just once and keep the position in a local variable? Such code would look like the following.

```
public void getReady()
{
    int myPosition = this.getPosition();
    while(myPosition < 6)

        this.jump(); this.right();
    }
    while (myPosition > 6)
    {
        this.jump(); this.left();
    }
    this.setColour(OUColour.RED);
}
```

Is your colleague's suggestion a good idea? If not, why not?

Solution

It's not a good idea at all! The `while` loops rely on checking the current position of the frog which, of course, changes as it moves. However, the value stored in `myPosition` is set at the start and does not get updated when the frog changes position. So the code above will not work.

The original version, which used `this.getPosition()`, is fine, because `getPostion()` always returns a message answer saying where the frog is at the present moment.

Although we have seen that it is preferable to make frogs responsible for their own sequence of movements, rather than having a `BarnDanceCaller` spell out each individual step, sticking to this policy can become difficult when the dances get very complex. This is the case in the next few activities, where we will see some fairly intricate interleaving of steps, and for simplicity we will therefore be specifying the details of each dance in the `BarnDanceCaller` class and not in the `Frog` class.

ACTIVITY 8

Open the project Unit7_Project_8.

This contains the code for the classes `Frog` and `BarnDanceCaller` as they were before the start of Activity 7.

You are now going to write a dance routine for a pair of frogs to perform. Remember, that before a dance commences the caller announces (metaphorically speaking) 'Get ready' and the pair of frogs which are to participate in the dance move to the central stones and turn red. When the dance finishes, the frogs metaphorically take off their red costumes and revert to their natural green colour.

In this dance, the frogs perform the following sequence of movements for a fixed number of times.

one step right, another step right, jump, then one step left

You will recall that you made a single frog move like this in *Unit 5*. However, in this case, the two frogs ***take turns to make each move*** in the above sequence.

Write a method of `BarnDanceCaller`, with the method heading

```
public void dance1(int aNumber)
```

which simulates the dance, and where the actual argument supplied is the number of times the sequence of movements is to be repeated.

Test that the frogs move as expected by executing a statement such as

```
caller.dance1(3);
```

in the OUWorkspace (remember to open the Graphical Display so that you can view the results).

DISCUSSION OF ACTIVITY 8

As usual, this is just one sample way that the method could be coded

```
   /**
    * Causes the dancers to first take up their positions, and then to
    * perform a dance sequence of right, right, jump, left which is
    * repeated aNumber of times before they turn back to green.
    * The dancers take turns to make each move
    */
   public void dance1(int aNumber)
   {
      this.takeUpYourPositions();
      for (int count = 1; count <= aNumber; count = count + 1)
      {
         this.getDancer1().right(); this.getDancer2().right();
         this.getDancer1().right(); this.getDancer2().right();
         this.getDancer1().jump(); this.getDancer2().jump();
         this.getDancer1().left(); this.getDancer2().left();
      }
      this.getDancer1().green();
      this.getDancer2().green();
   }
```

Exercise 6

Even though the activities of the two frogs are now thoroughly interleaved, the above method involves some repetition. Suggest a useful refactoring that could be made to mitigate the repetition of sending `right()` messages to the dancers.

Solution

One possible refactoring might involve constructing a new helper method of `BarnDanceCaller` as follows

```
   private void bothRightTwice()
   {
      this.getDancer1().right(); this.getDancer2().right();
      this.getDancer1().right(); this.getDancer2().right();
   }
```

However, such a method would still involve exactly the same repetition, we have merely moved it elsewhere (unless some other dance uses exactly the same move). Consequently, a better answer would probably be to suggest a new method of `BarnDanceCaller` as follows (using Design principle 1).

```
   private void bothRight()
   {
      this.getDancer1().right(); this.getDancer2().right();
   }
```

This would then allow the method `dance1()` to be refactored as follows

```
public void dance1(int aNumber)
{
    this.takeUpYourPositions();
    for (int count = 1; count <= aNumber; count = count + 1)
    {
        this.bothRight(); this.bothRight();
        this.getDancer1().jump(); this.getDancer2().jump();
        this.getDancer1().left(); this.getDancer2().left();

    this.getDancer1().green();
    this.getDancer2().green();
}
```

This possible refactoring has obvious repetition in the line

```
this.bothRight(); this.bothRight();
```

so further refactoring would be possible by the introduction of yet another helper method, a new version of the `bothRightTwice()` method

```
private void bothRightTwice()
{
    this.bothRight(); this.bothRight();
}
```

Consequently, the method `dance1()` could now be refactored as follows:

```
public void dance1(int aNumber)
{
    this.takeUpYourPositions();
    for (int count = 1; count <= aNumber; count = count + 1)

        [illegible]
        this.getDancer1().jump(); this.getDancer2().jump();
        this.getDancer1().left(); this.getDancer2().left();
    }
    this.getDancer1().green();
    this.getDancer2().green();
}
```

The suggested refactoring of `dance1()` in the above exercise is extremely instructive, because it now contains statements close together that work at very different 'levels of detail', as follows

```
this.bothRightTwice();
this.getDancer1().jump();
```

Design principle 3

Wherever possible the actions in a single method should be at a single level of detail. Actions that must necessarily be at different levels of detail are better put in different methods, where possible.

This idea of 'levels of detail' is necessarily a little vague, but one crude way of pinning down the different levels of detail involved in this particular example is to notice that the first statement causes four frog movements, whereas the second statement causes a single frog movement.

There is a general design principle (Design principle 3) that says that in general, the actions in a single method should be at a single level of detail, where possible. Actions that must necessarily be at different levels of detail are better put in different methods where feasible

One way to apply this principle in the above example would be to create more helper methods. One good candidate would be as follows:

```
private void bothJump()
{
   this.getDancer1().jump(); this.getDancer2().jump();
```

Exercise 7

Suggest two more methods that could be defined in a similar spirit to [illegible] as a way of further applying Design principle 3 to refactoring `dance1()`.

Solution

Using Design principle 3 to guide further refactoring might lead to the following helper methods (other variants are possible)

```
private void bothLeft()

   this.getDancer1().left(); this.getDancer2().left();
}
private void bothGreen()

   this.getDancer1().green(); this.getDancer2().green();
```

This would then give us the following refactored `dance1()` method:

```
private void dance1(int aNumber)
{
   this.takeUpYourPositions();
   for (int count = 1; count <= aNumber; count = count + 1)
   {
      this.bothRightTwice();
      this.bothJump();
      this.bothLeft();
   }
```

Applying Design principles 1 and 3 as we did in the last two exercises would make the `BarnDanceCaller` class clearer and easier to read, and more maintainable.

However, for teaching reasons, in the next activity, using the `BarnDanceCaller` class, we will temporarily ignore these improvements and so will take the starting point of the code for the next activity as it was at the end of Activity 8.

ACTIVITY 9

Open the project Unit7_Project_9.

In this activity you are going to write a method to orchestrate another dance. In this dance, the frogs perform the same sequence of movements as they did in the first dance, except that the dance finishes when both dancers reach a particular specified stone.

Write a method of `BarnDanceCaller`, with the method heading

```
public void dance2(int aPosition)
```

which simulates the second dance, and where the actual argument supplied is the stone (i.e. position) where the dance finishes.

In this dance routine, the net result of the movements is that the dancers move progressively to the right controlled by a while loop. So if the actual argument supplied to `dance2()` represents a stone which is to the *left* of the central stone, the dancers should not move after `takeUpYourPositions()` has turned them red and moved them to the central stone (they should, however, revert back to their natural colour of green).

Test that your method works using message-sends such as `caller.dance2(8)` and `caller.dance2(4)`. But beware! If your method omits a test for the finish position (that is, the stone where the dance ends), be prepared for the frogs trying to dance forever. You will need to remember the advice given in *Unit 5* and the *Software Guide* about what to do in such a situation.

DISCUSSION OF ACTIVITY 9

The trickiest part of this method is the condition controlling repetition of the rightward dances. The dances need to be repeated until both dancers have reached `aPosition`. The following condition ought to evaluate to `true` until *both* dancers have reached the final position `aPosition`.

```
(dancer1.getPosition() < aPosition) || (dancer2.getPosition() < aPosition)
```

In the following method a `while` loop statement contains a combined condition to check that both dancers have not yet reached the specified stone. The frogs will continue to dance while the evaluation of this combined condition above results in `true`.

```
/**
 * Causes the dancers to move as in dance1, except they stop when
 * a given stone, represented by the argument aPosition is reached.
 * If the argument supplied represents a stone which is to the left
 * of the central stone, the dancers do not move after takeUpYourPositions()
 * has turned them red and moved them to the central stone, however they do
 * revert back to their natural colour of green.
 */
public void dance2(int aPosition)

   this.takeUpYourPositions();
   while ((this.getDancer1().getPosition() < aPosition)
            this.getDancer2().getPosition() < aPosition))
   {
      this.getDancer1().right(); this.getDancer2().right();
      this.getDancer1().right(); this.getDancer2().right();
      this.getDancer1().jump(); this.getDancer2().jump();
      this.getDancer1().left(); this.getDancer2().left();
   }
   this.getDancer1().green();
   this.getDancer2().green();
```

Note that this implementation assumes that the `takeUpYourPositions()` message will result in both dancers being in the same starting position, namely position 6

Note also that the requirement for the dance only to take place while the dancers are to the left of `aPosition` is implemented by the tests involving the operator <. If that test were to be omitted, the dancers would potentially continue dancing forever.

Of course, the code above could be greatly improved by judicious refactoring, but we will not explore this

The code for the `dance2()` *method has been added to the* `BarnDanceCaller` *class in Unit7_Project_10*

ACTIVITY 10

The third and final dance is exactly the same as the first dance except that when the frogs have finished dancing, they return directly to the stones they occupied before the caller announced 'Get ready' with a `takeUpYourPositions()` message.

Open the project Unit7_Project_10.

Write a method of `BarnDanceCaller`, with the method heading

```
public void dance3(int aNumber)
```

The method will be identical to `dance1()` except that your code will need to remember the dancers' starting positions and return them to those positions once the dance is over Hence you should make use of a `dance1()` message in your method code.

In the OUWorkspace test that your method works correctly When testing, start your two frogs from two different positions.

DISCUSSION OF ACTIVITY 10

In the following method the initial positions of the participating frogs (part of their state) are preserved by using two local variables, `position1` and `position2` After the dance has finished, the frogs are restored to their initial positions You can cause the frogs to perform the first dance simply by sending the message `dance1()` to the receiver, the orchestrating instance

```
/**
 * Cause the frog objects identified by dancer1 and dancer2 to
 * perform dance1 aNumber of times and then return directly to the
 * stones they occupied before the dance began
 */
public void dance3(int aNumber)
{
    // Save starting positions.
    int position1 = this.getDancer1().getPosition();
    int position2 = this.getDancer2().getPosition();

    // Execute the dance
    this.dance1(aNumber);

    // Restore both positions
    this.getDancer1().setPosition(position1);
    this.getDancer2().setPosition(position2);
}
```

As before, this method offers several opportunities for refactoring.

The code for the `dance3()` *method has been added to the* `BarnDanceCaller` *class in Unit7_Project_10_sol.*

2.4 Simulation and reality

The previous activities illustrate some of the differences between an aspect of barn dancing in the real world and using a computer system to simulate that aspect. You need to understand the implications of the differences, as outlined below.

In the simulation of barn dancing you carried out in the activities, the orchestrating instance (representing the caller in the real world) needs to send a message to each individual dancer object in turn to instruct them all to perform the next dance movement. Before the orchestrating instance can send these messages to the dancers, it needs to know where to find them; that is, it needs to hold a reference to each dancer object.

In the real physical world, the dancers and the caller would presumably be in the same room, or at least in earshot of each other, so the caller would simply have to call out each dance movement once, and the instruction would be received by all the individuals participating in the dance. But there is no simple way of mirroring this directly with objects. That is to say, there is no simple way of specifying that a set of objects are all 'in earshot' of each other (though an important partial approximation to this idea will be introduced in *Unit 10*).

Consequently, in your code, individual messages have to be sent to each dancer in the simulation of barn dancing. This is perhaps as though real world dancers each had to be telephoned individually to pass on each command. As a result, dancers perform each dance movement one after another rather than concurrently as they would in real life.

SAQ 9

Describe how an orchestrating instance carries out its role as a coordinator of other objects' actions.

ANSWER.......

The orchestrating instance holds references to the other objects and sends them messages to coordinate their actions.

SAQ 10

Describe how you would modify the `BarnDanceCaller` class so that an instance could coordinate four rather two frogs to perform the same dances.

ANSWER

Add two more instance variables to hold two more frogs, and adapt the methods accordingly. (After you have studied *Unit 10*, you may be able to see other or better solutions.)

3 Class methods and class variables

You were introduced to class methods in Section 1 of *Unit 5*. The various methods of the class `OUDialog`, such as `alert()` and `request()` are all class methods and you have used them in statements such as

```
OUDialog.alert("Illegal input");
```

Similarly, you have been using class variables since Unit 3: the various predefined colours that you have used with amphibians, such as `GREEN` and `RED` are all class variables of the `OUColour` class which you have used in statements such as

```
frog1.setColour(OUColour.GREEN);
```

3.1 Class variables

You are already familiar with instance variables in Java. Recall that instance variables are defined by the class, but each object carries around its own unique copy of the instance variables for its class, and uses these to store information about its individual state, independently from any other object of the same class.

As well as instance variables, Java has what are known as **class variables**. In contrast to instance variables, there is only a single copy of a class variable and it belongs to the class as a whole. A class variable does not in general depend on any objects of the class having been created, and is available for use immediately the class is first used. Indeed, normally speaking, if we were to make it so that no objects of that class ever came into existence, the class variable would still be there.

What sort of things would we want a class variable for? Here is an example of one possible use. Suppose there is a kind of frog, called a bovverfrog, which is highly territorial and extremely touchy about sharing stones with other bovverfrogs (when it is first created). Therefore when a `BovverFrog` object is constructed, it has to be placed on a different initial stone from any of the previous bovverfrogs. There could be many ways to ensure this, but one very simple means is to put the first bovverfrog on stone 1, the second on stone 2, and so on.

To put this idea into practice, we would need some way of keeping track of which stone the next bovverfrog should go on. Using an instance variable to record the number would be a possible solution, but not a very good one. Every bovverfrog would have its own distinct copy of the instance variable. So we would have either to keep all of them updated – unnecessarily complicated to code and a waste of effort and computer memory – or to pick on a particular bovverfrog, make it solely responsible, and ignore all the other copies of the instance variable – but how would we decide to favour one bovverfrog over all the others (especially given their touchy nature!)? In both cases, we would be allocating space for storing lots of numbers, when space for a single number is all that is actually required.

Instead, a far better solution is to use a class variable called, say, `nextEmptyStone`. There is a single copy of this for the whole class. It is initialised to 1 when the `BovverFrog` class is first used. Then every time a new `BovverFrog` object is constructed it is placed on `nextEmptyStone`, then `nextEmptyStone` is increased by one by the constructor, ready for the next bovverfrog.

Declaration and initialisation of class variables

Here is the relevant part of the class definition that declares and initialises the class variable `nextEmptyStone`.

```
public class BovverFrog extends Frog
{
    public static int nextEmptyStone = 1;
    ...
     .
}
```

Notice that the class variable declaration is done right at the beginning of a class definition. This is where class variable declarations should typically appear. (It is not forbidden for them to be declared elsewhere in the class, provided they are not inside a method or constructor, but the common convention is for them to go where we suggest, which makes them easy to find.)

Note how the keyword `static` is used in the variable declaration to show that this is a *class* variable, not an instance variable.

Notice too that the class variable has been declared as public; this means that any object of any class will be able to directly access the class variable. If we had declared the variable as private, only class methods, instance methods or constructors of the `BovverFrog` class would be able to directly access it.

Note also that the declaration also initialises the class variable to `1`. There is an interesting contrast between the way that a class variable such as `nextEmptyStone` is initialised (i.e. given an initial value), and the way in which instance variables such as `colour` and `position` are initialised. Instance variables have to be initialised separately for every new instance, and this is reflected by the fact that they are usually initialised in a constructor. However, class variables only have to be initialised once, when the program first uses the class. To reflect this difference, a class variable is initialised within the class definition, but outside any methods and constructor definitions. Class variables are typically declared and initialised all in one statement, for example:

```
public static int nextEmptyStone = 1;
```

You might think that it would work equally well to separate the declaration of a class variable and the assignment of its value into two statements as follows:

```
public static int nextEmptyStone;
BovverFrog.nextEmptyStone = 0;
```

However, this is not legal in Java and will not compile. There is a way that the statements can be separated, if they are enclosed in what is called a static block, but this takes us beyond the scope of the course.

Accessing a class variable

In our example, the constructor for instances of the `BovverFrog` class directly accesses the `nextEmptyStone` to increment it whenever a bovverfrog is created, so ensuring that no two bovverfrogs are initially put on the same stone.

```
public BovverFrog()
{
    super();
    // Place the bovverfrog on the next empty stone.
    this.setPosition(BovverFrog.nextEmptyStone);
    // Increment nextEmptyStone
    BovverFrog.nextEmptyStone = BovverFrog.nextEmptyStone + 1;
```

Notice how in the above code [illegible] is **qualified** with the class name `BovverFrog`. This is not strictly necessary within class methods, instance methods or constructors of the class that defines the class variable, but it is good programming style and complements the use of `this` to make it clear to the reader of the code that `nextEmptyStone` is a class variable and not an instance variable. Objects of other classes that might want to access `nextEmptyStone` *must* qualify the variable with the class name.

The next activity is optional but demonstrates the effect of the constructor incrementing the class variable `nextEmptyStone`.

ACTIVITY 11

Open Unit7_Project_11.

This includes the [illegible] class. Open the OUWorkspace and the Graphical Display.

Now enter and execute the following statements one by one.

```
BovverFrog sid = new BovverFrog();
BovverFrog em = new BovverFrog();
BovverFrog daf = new BovverFrog();
```

DISCUSSION OF ACTIVITY 11

You should have seen that each successive bovverfrog is placed on its own stone, as required to avoid any unpleasant interaction between the bovverfrogs.

3.2 Class methods

As well as class variables, Java has **class methods**. These can be used irrespective of whether any instances of the class have been created, and they are executed by invoking them not on an object, but on the name of the class itself. The statements look a bit as if we are sending a message to the class, and you might well find it helpful to think of it that way although strictly speaking this is not what is happening, since in Java a class is not an object.

Returning to our `BovverFrog` class, consider a class method called `isNextEmptyStoneBlack()` which would return `true` or `false` depending on whether the next bovverfrog will be placed on one of the black stones in the Amphibians window. Static method definitions are easier to find if they are put together near the beginning of the class file, before any instance method definitions but after any constructors. Here is how the class method for `BovverFrog` is declared; we have omitted the comments for brevity.

```
public static boolean isNextEmptyStoneBlack()
{
   return BovverFrog.nextEmptyStone <= 11;
}
```

Notice how, just like class variables, class methods are declared using the keyword `static`.

The method can be invoked on the class as follows:

```
BovverFrog.isNextEmptyStoneBlack();
```

Invocation of class methods

There is more than one way to invoke a class method such as `isNextEmptyStoneBlack()`. The obvious way, which you have already seen, is as follows

```
BovverFrog.isNextEmptyStoneBlack();
```

This makes it immediately clear to the reader that a class method is being invoked. Of course, as `isNextEmptyStoneBlack()` is a `public` class method, this would work from a method of any class in the same package, or from any class that imported `BovverFrog`.

However Java allows you to write code to invoke a class method which *looks* like you are invoking a class method on an instance of a class, for example the code

```
BovverFrog grumpy = new BovverFrog();
grumpy.isNextEmptyStoneBlack();
```

will invoke the `BovverFrog` class method `isNextEmptyStoneBlack()`. In fact the Java compiler produces bytecode which is the equivalent of

```
BovverFrog.isNextEmptyStoneBlack();
```

and it ignores the class of the object referenced by `grumpy`. Instead, the compiler looks at the *declared type* of the variable (`grumpy`), and uses that declared type to determine, *at compile time*, which method to call. Since `grumpy` is declared as type `BovverFrog` the compiler looks at the code `grumpy.isNextEmptyStoneBlack()` and decides it actually means `BovverFrog.isNextEmptyStoneBlack()`. It doesn't matter that the object referenced by `grumpy` is an instance of `BovverFrog` as for static methods, the compiler only uses the declared type of the reference.

For this reason this style of class method invocation is considered bad practice, as it makes it look to the reader as if `isNextEmptyStoneBlack()` is a message that will lead to the invocation of an instance method at run-time, although in fact it is an invocation of a class method.

Inheritance vs. visibility of class methods and class variables in subclasses

Classes are not objects in Java, therefore inheritance of class methods and class variables is *not* possible in the same way that instance methods and variables are inherited. However Java does provide an *approximation* of inheritance for class methods and variables, in that if they are declared as `public` or `protected` they are *visible* from within the methods of any subclass. Furthermore, you can also use the subclass name to qualify the class method or class variable. For example, given a hypothetical class called `SuperClass` that defines a class method called `foo()`, and a subclass of `SuperClass` called `SubClass`, then you can write code that looks like the class method `foo()` is being invoked on the subclass as follows. `SubClass.foo();` However, once again, the Java compiler produces bytecode which is the equivalent of

```
SuperClass.foo();
```

For many everyday purposes this kind of behaviour looks much like inheritance, but there are crucial differences that can lead to unexpected problems. For example problems can arise if you *try* to override a class method in a subclass. The compiler will allow you to do this, but the method isn't really overridden. Instead of overriding, the result is something called **hiding** or **masking**, whereby the method in the superclass is hidden from the subclass. This can lead to highly confusing results in certain circumstances, for example if the programmer writes code that looks as if it is invoking a class method on an instance of a class.

It is beyond the scope of this course to map out these differences, but we can give a simple set of guidelines for avoiding the resulting problems in the first place. Provided you

observe these guidelines you may, if you wish, think of class methods and class variables as being inherited, though you should be aware that in a strict sense this is misleading.

Our advice to you when using class methods and class variables, in Java, is as follows.

- Only invoke class methods on the name of the class in which they are defined, never on a subclass and never on the name of a variable.
- Only access class variables by qualifying them with the name of the class in which they are declared, never qualify them with the name of a subclass or the name of a variable.
- Always give class methods and class variables names which are different from any other names used in the class or any of its superclasses.

If you follow this advice you won't go wrong with the useful fiction that class methods and class variables in Java are inherited. Stray from this and you will find that your code will exhibit behaviour that can be very confusing and hard to debug.

ACTIVITY 12

The class method [illegible] has been added to the [illegible] class in Unit7_Project_12, and if you open that project the method can be tested in the OUWorkspace. Open the Graphical Display and then execute:

```
BovverFrog.isNextEmptyStoneBlack();
```

Note the message answer displayed in the Display Pane. Now create eleven bovverfrogs, like this:

```
for (int count = 1; count <= 11; count++)
{
   new BovverFrog();
}
```

Notice we are not bothering to assign the [illegible] objects to variables, since we do not want to use them again, there is no need. We are simply creating them to push the next free stone off to the right. Because none of them are anchored by a reference, the [illegible] objects are garbage collected immediately, and do not show up in the Amphibians window.

Execute `BovverFrog.isNextEmptyStoneBlack();` again and note the message answer shown in the Display Pane. You can confirm whether this message answer is indeed correct by creating one more bovverfrog, assigning it to a variable like this:

```
BovverFrog tel = new BovverFrog();
```

and noting the colour of the stone on which it appears in the Amphibians window (you may have to scroll this window to the right to see the bovverfrog.

DISCUSSION OF ACTIVITY 12

The first time you sent the `isNextEmptyStoneBlack()` message to the `BovverFrog` class, the message answer should have been `true`, then after creating the eleven bovverfrogs the message `isNextEmptyStoneBlack()` should have returned `false`. Finally, when you created a twelfth bovverfrog and assigned it to `tel`, the bovverfrog should have appeared on the first blue stone in the Amphibians window.

As a minor point about terminology, now that you have been introduced to class methods and class variables, it is worth mentioning in passing that the term **member** is sometimes used to cover all of the four following categories: class variables, class methods, instance variables and instance methods.

3.3 Adding a class variable and class method to the `Frog` class

In this subsection, you will add new class variables and class methods to the `Frog` class. As a simple example, we will use a class variable `frogCount` to count the number of instances of `Frog` that are created

ACTIVITY 13

Open the project Unit7_Project_13, then open the editor on the `Frog` class.

After the first line defining the class and immediately after its opening curly bracket start a new line to declare a class variable `frogCount` of type `int`. You should declare this class variable to be `private`, and initialise its value to `0`.

Next, after the constructor, create a class method with the heading.

```
public static int getFrogCount()
```

which should simply return `frogCount`.

Do not forget that both variable and method must be declared `static`.

Compile your modified `Frog` class.

Now in the OUWorkspace, try directly accessing by executing the following statement:

```
Frog.frogCount;
```

What happens?

Now try invoking the class method on the class by executing the following statement:

```
Frog.getFrogCount();
```

What happens?

DISCUSSION OF ACTIVITY 13

The declaration of the class variable `frogCount` should look something like the following

```
private static int frogCount = 0;
```

Your class method `getFrogCount()` should look like the following

```
public static int getFrogCount()
{
   return Frog.frogCount;
}
```

Notice that `frogCount` is declared `private` and so any attempt to access the class variable `frogCount` directly by executing the statement `Frog.frogCount` in the OUWorkspace will fail, since you would be trying to access a private member from outside the class or its instances (namely you are trying to access it from the OUWorkspace). A typical error message given in the Display Pane would be as follows

```
Semantic error: No static field or inner class: frogCount of class Frog
```

However, the *class method* `getFrogCount()` is `public`, so the invocation of the class method `getFrogCount()` on the class `Frog` is successful Executing the statement

```
Frog.getFrogCount();
```

returns `0`.

Now that you have successfully defined the class variable `frogCount` and the asscociated getter method the next step is to use it to count the number of instances of `Frog` that are created

ACTIVITY 14

Open Unit7 Project 14 which includes the class variable [illegible] and the class method [illegible] in the [illegible] class Open the OUWorkspace and the Graphical Display Modify the constructor [illegible] so that the last thing it does is to increment the class variable [illegible] by [illegible] via direct access (remember the variable will be accessible *from within the class*, even though it is `private`).

When you have compiled your new version of the [illegible] class in the OUWorkspace create a frog referenced by the variable [illegible] Then invoke the class method on the [illegible] class to check the new value of [illegible] Finally create three more frogs but this time do not create any references to these frogs (That is to say, you should use [illegible] to create them, and the constructor to initialise them, but do not assign them to any variable allowing them to be garbage collected) Then check the value of [illegible] again

DISCUSSION OF ACTIVITY 14

In the constructor `Frog()` we can increment `frogCount` with the statement

```
Frog.frogCount = Frog.frogCount + 1;
```

Once the `Frog` class has been recompiled with this alteration, if you create a new instance of `Frog` referenced by `jimi`, as follows

```
Frog jimi = new Frog();
```

then when you execute

```
Frog.getFrogCount();
```

the return value should now be `1`.

The counting works just as well when you try the following

```
for (int i = 0; i < 1000; i++)
{
   new Frog();
}
```

After this

```
Frog.getFrogCount()
```

will return `1001`

This is not the way that you would normally create instances of `Frog` – creating a frog without assigning the new frog to a variable means that there is no reference to the newly created object and the new instance will be lost. However, in this case it does not matter that the new frog instances are lost because you are creating frogs just to count them What you are really doing is counting the number of times the constructor `Frog()` is run – but that is just what we want

The class variable `frogCount` was intended to hold a count of instances of the `Frog` class. In the next activity you will explore what happens when new instances of `HoverFrog` are created.

ACTIVITY 15

Open the project Unit7_Project_15.

This includes the modified constructor for the [illegible] class. Then in the OUWorkspace invoke the `getFrogCount()` class method on the `Frog` class to check the value of class variable `frogCount`.

Create a frog and then check the value of `frogCount` again.

Create a hoverfrog and check the value of `frogCount` once more.

DISCUSSION OF ACTIVITY 15

Initially the count should be zero. Now suppose we execute the following code

```
Frog fry = new Frog();
HoverFrog hal = new HoverFrog();
```

Checking the value of `frogCount` now gives the value `2`.

```
Frog.getFrogCount();
```

In the previous activity you may have been surprised to discover that creating a hoverfrog resulted in the class variable `frogCount` of `Frog` being incremented – after all `frogCount` was declared as `private` and therefore it is not directly accessible by the `HoverFrog` class. So why has this happened?

The reason is that constructors are **chained**. When an object is created, the constructors of its superclasses are always called. So, in the process of constructing a `HoverFrog` object, Java will invoke the constructor from `Frog`.

Depending how you write the code, this invocation might be *explicit* – the first line of the `HoverFrog` constructor is `super()` – or it might be *implicit*, supplied by Java behind the scenes. Even if you do not write any constructor at all for `HoverFrog`, the rules of Java dictate that one will be inserted automatically in the compiled code, and this supplied constructor will still invoke the constructor `Frog`.

So whatever you do, in the process of creating a `HoverFrog` the constructor `Frog()` will be executed and *it will increment the* `frogCount`.

For some purposes it is useful for classes and subclasses to share information in this way – for example, for one purpose we might decide that hoverfrogs are a kind of frog, so we would be happy for each new hoverfrog to be counted as a frog. However, for some other purposes, we might not want to have `HoverFrog` instances affect the value of `frogCount` – we might want the hoverfrogs to have their own separate count of instances. How could we achieve this? We will consider one approach in the next subsection.

Polymorphism, instance methods and class variables

If we require different behaviour depending on the class of an object the best approach is almost always to involve instance methods in the solution, because different classes can provide different implementations of the same method signature. Hence the corresponding message becomes polymorphic – objects of different classes will

respond differently to the message. The clever idea in this case is that we continue to use the class variable and method to keep count, but use an instance method to decide whether the count should be incremented or not.

ACTIVITY 16

Open the project Unit7_Project_16.

Give `Frog` a new class method `incrementFrogCount()`, with a `void` return value, which directly increments `frogCount` by `1`. Give `Frog` a new *instance* method `incrementInstanceCount()`, with no argument or return value, whose effect is to invoke the *class* method `incrementFrogCount()` on the `Frog` class.

Alter the [illegible] constructor [illegible] so that it no longer directly increments the class variable `frogCount`. Instead, have it send the message `incrementInstanceCount()` to the newly created object, which can be referenced by [illegible] from within the constructor.

In `HoverFrog`, override the inherited *instance* method `incrementInstanceCount()` to do nothing, i.e. the method should have no code within the enclosing braces { } of the method.

In the OUWorkspace, execute the following statements. What happens this time?

```
Frog fry = new Frog();
HoverFrog hal = new HoverFrog();
Frog.getFrogCount();
HoverFrog.getFrogCount();
```

DISCUSSION OF ACTIVITY 16

Now checking the value of `frogCount` reveals that `frogCount` is only incremented when `Frog` instances are created, not when `Hoverfrog` instances are created.

Of course, you could go further than this by [illegible] own separate count of hoverfrogs. This could be done by creating in `HoverFrog` a new class variable `hoverFrogCount`, and a new class method `getHoverFrogCount()` together with a new class method `incrementHoverFrogCount()` that increments `hoverFrogCount` by 1. Then you would need only to modify the instance method `incrementInstanceCount()` in `HoverFrog` to invoke `incrementHoverFrogCount()` on `HoverFrog` instead of just doing nothing.

You can find the completed code for this activity in the project Unit7_Project_16_sol.

We shall now go on to explore some other examples of class variables and methods, beginning with an example using class 'variables' which do not actually vary[1].

3.4 Constants

A **constant** is a variable whose value is fixed and unchangeable. For example, the number of legs on a spider is eight, so that if we wish to model a collection of intact, well-formed spiders, they will all, without exception, have eight legs. To represent this sort of unvarying attribute or information, we use a class variable and add the Java keyword `final`, which prevents anyone ever assigning a new value to it. By convention – although Java does not enforce this – constants are usually written using upper case and the underscore character, as in the following example

```
public class LegsKnowledgeBase
{
    public static final int SPIDER_LEGS = 8;
    public static final int INSECT_LEGS = 6;
    public static final int HUMAN_LEGS = 2;
}
```

Notice that the code above constitutes an entire class and its sole purpose is to perform a public service for objects of other classes – to make available to them information about how many legs spiders, insects and humans have. If an object of another class needed to know the information it would use the name of the class where the constant is declared and the name of the constant like the following

```
int legs = LegsKnowledgeBase.SPIDER_LEGS;
```

Now `legs` will hold the value 8. Notice that this is not a method invocation – there are no parentheses – instead the constant is directly accessed by its name prefixed by the name of the class it is defined in, with a dot between the two.

Of course, we do not have to do things this way, with a special `LegsKnowledgeBase` class. For example, if there was a `Spider` class, we could have defined a class variable within the `Spider` class instead – and for many purposes, this might be the neatest solution. Still, both approaches are used, and both can have benefits. By contrast, we could do things without using a class variable at all. Each object that needed to could remember the facts about spiders itself. But this instance variable based approach has big disadvantages, which you may remember from the discussion of bovverfrogs.

SAQ 11

What would be the disadvantages of an instance variable based approach to storing a fact about spiders that we are certain (for the purposes of some model) will never vary?

ANSWER

- Storage is wasted by the duplication.
- More importantly, the probability of error and inconsistency is increased. A programming error might cause one or more copies of the instance variable to have the wrong value.

Describing constants as one kind of variable may sound slightly strange from a common sense point of view. After all, in ordinary English, variables and constants are quite different things. However, in Java, constants are implemented as variables which happen to have been declared as `final` to stop them varying. Thus, for Java purposes, it makes perfect sense to think about constants as one kind of variable, otherwise many general statements about Java variables would become messily cluttered by having to mention constants specially.

SAQ 12

If we wanted to expand the legs knowledge base to include starfish (5-legged ones), what extra line would we need to add to the class `LegsKnowledgeBase`?

ANSWER

```
public static final int STARFISH_LEGS = 5;
```

We have been focusing on examples of class variables that happen to be declared as constants (using the keywords `static final`), but it is also perfectly possible to make an instance variable constant in a similar way, simply by marking it as `final`. However, there is an important difference between a class constant and an instance constant. A constant class variable has the same value for *all* instances of a class, whereas a constant instance variable may well have different values for different instances. For example, a **constant instance variable** could be declared and used to give each `Account` a different `accountNumber`. This would have its value set when the account was initialised, in the constructor, and could never be changed thereafter.

4 A review of the different kinds of variable

In this section we review the different kinds of variables used in the course

All types of variable have the same purpose – to provide a name for a location which holds a value, which can be either a primitive data value or a reference to an object. Variables all work in essentially the same way but they differ in how and where they can be used. You have used four **sorts of variable** so far (all of which can hold either kind of value).

4.1 Local variables

Local variables are declared for the purposes of a single method, constructor or block, which is where their declarations are found. A new copy of a local variable is created each time the method, constructor or block is executed, and is destroyed as soon as execution is completed. Local variables are not accessible outside of the relevant method, constructor or block in which they are declared.

4.2 Workspace variables

If a Java development environment provides a workspace, like the one in M255, there [illegible] **workspace variables**. [illegible] behave like local variables except that their lifetime lasts until you close or reset the workspace, rather than ending when a particular block has finished executing.

4.3 Instance variables

Unlike local variables, instance variables are not declared within any method, constructor or block, but as separate items in the class definition.

In terms of their *usage*, instance variables contrast even more sharply with local variables. Instance variables are used in an object to refer to other objects or to hold primitive values that make up its state. There is an individual copy of each instance variable per instance of the class – each instance may have a different value for the same instance variable compared with the other instances of the same class. An instance variable is created and destroyed at the same time as the instance that contains it. While its value may change during the lifetime of the object, the instance variable itself has the same lifetime as the containing object – no shorter and no longer. Notice that when an instance variable is destroyed, any object that it refers to may or may not be destroyed – that depends on whether one or more other references to that object still exist.

`private` is the most restrictive level of access. Instance variables may be given more visibility by defining their access as `package`, `protected` or `public`, in order of widening permissiveness.

Even if an instance variable is private, it can still be directly accessed from any instance method or constructor of that class. Much more surprisingly, if we have two instances of the same class, and one has a reference to the other, then it will be able to directly access the other's instance variables, even if they are private. In other words, `private` indicates that the instance variable is private to all instances of that class; instances of other classes cannot access it directly, even an instance of a subclass that inherits a private instance variable will be unable to access the inherited instance variable directly.

Normally you should expect instance variables to have setter and getter methods, so that the instance variable is always set with a message-send such as

```
someObj.setSpeed(90);
```

or

```
this.setspeed(90);
```

rather than directly with a statement such as

```
myObj.speed = 90;
```

or

```
this.speed = 90;
```

Accessing a variable directly from a method is perfectly reasonable in small example programs, but can be a dangerous habit in larger programs where many methods need to access the same variable. The problem with direct access will arise if the program grows and ever needs to be modified at some point in the future (for example, to make some action occur whenever the value of variable is altered or accessed, or to change the way in which it is stored). If the variable is accessed directly, then there will be no alternative but to make changes in every method where the access takes place. This is often laborious, and can be downright dangerous if one or more places are missed.

On the other hand, if the discipline has been adopted right from the start of accessing variables only via their getter and setter methods, then any needed changes can be limited to the setter and getter, however many methods may use the setter and getter. This is why this course tends to discourage **direct access to instance variables** from methods, except by at most one method that sets the value directly by assignment, and at most one method that returns its value. Recall, however, that in the case of *constructors*, direct setting of variables is preferred.

In fairness, to put the opposite view, in the case of very simple classes that are unlikely to change, the increased flexibility comes at the cost of having to write the setters and getters, and having code that you may consider takes marginally more effort to read and write.

Hence, there is a trade-off between flexibility and directness. In different contexts, different conventions are legitimately preferred. However, for the purposes of this course we will expect getters and setters to be used as outlined above.

Inheritance vs. visibility

It is important to distinguish between inheritance and visibility. If an instance variable has been declared as `private`, then it will not be directly accessible by instances of any subclasses. However, this does not mean that the instance variable is not present in instances of the subclass. This might sound like a meaningless distinction, but, in fact, it is a very concrete issue. Subclasses *always* inherit any instance variables defined in their superclasses (as can be proved by inspecting a hoverfrog in the OUWorkspace); it is just that instances of a subclass cannot 'see' inherited instance variables that have been declared as `private`. However, provided there are public or protected getter and setter methods for the instance variable defined in the same class as the variable, then these methods will be inherited by the subclass, providing indirect access to the instance variable from instances of the subclass. This may sound somewhat roundabout, but it can be very useful, as it could allow the original class to change the way it stores the instance variable without inconveniencing any subclasses which access this state via the inherited getter.

There are two very common ways to become confused about the absence of instance variables that you are convinced should be present. This typically happens when creating a new class or when using a class for the first time. First, since instance variables belong to instances, there are *no* copies of the instance variables until an instance of the class is actually created. Second, since a static method can be executed *before* any instances exist, it follows that it is not possible for it to make use of instance variables – *unless*, of course, the static method first creates an instance of the class and then it can access the copy of the instance variables belonging to the instance created.

4.4 Class variables

Class variables [illegible] **static variables** [illegible] keyword `static`. Class variables represent the state associated with a class irrespective of whether any instances have been created, and irrespective of the state of any instances that have been created. There is just one copy of each class variable per class. Class variables, like instance variables, are declared as distinct items within the class definition. They are usually initialised where they are declared.

A class variable is initialised at first use in a particular run of a program, and is destroyed only when program execution ceases. This contrasts with an instance variable, which exists only as long as something is referencing the object it belongs to.

Regardless of whether a class variable is private, any instance method or constructor can access any class variable declared in its class. Class variables may be given more **visibility** by defining them using a more permissive level of access modifier than `private`. Like instance variables, they may have `package` (the default), `protected`, `private` or `public` accessibility. For example, if a class variable is defined using the modifier `protected`, then subclasses and instances of subclasses will be able to access this class variable, as well as other classes in the same package. In this course we only make use of public and private accessibility.

To access a class variable from outside its class, its name must be **qualified** by the name of its class. To access it from a different package we must qualify the name by the class *and* the package. So if we had a package `marine` containing a class `Octopus` with a `public` class variable `noOfLegs`, it would be accessed from within the `Octopus` class and its subclasses by the name `noOfLegs`, from within the package `marine` by the name qualified by class, `Octopus.noOfLegs`, and from outside the package by the doubly qualified name `marine.Octopus.noOfLegs`.

The following tables summarise the different kinds of variables.

Variable type	Where declared	Where accessible	When created	When destroyed
Local	In a method constructor or block.	Within the method constructor or block in which declared.	Each time the method constructor or block is executed.	When the execution of method, constructor or block is completed
Workspace	In the workspace	From any statement executed in the workspace.	When the variable declaration is executed in the workspace.	When the workspace is closed or reset
Instance	Within the class definition (but *not* within any method constructor or block).	See the next table	When the object in which it is contained is created.	When the object in which it is contained is garbage collected because it no longer has a reference
Class	Within a class definition (but not within any method, constructor or block) and declared as `static`	See the next table	When the class is first used in an executing statement in the workspace or any other running program.	When the workspace is reset or closed (more generally when the program in which it is used ends) When the class is re-compiled

	Where accessible
Instance	1 From any instance method or constructor of the containing object (via [illegible] and irrespective of the privacy of the instance variable). 2 From any other object of the *same* class, provided that object has a reference to the object containing the instance variable and irrespective of the privacy of the instance variable. Instance variables may be given more visibility by using the access modifiers [illegible] and `public`. *Inadvertent effects* May be hidden in an instance of the subclass by an instance variable of the same name declared in the subclass
Class	1 From any class method, instance method or constructor of the defining class, irrespective of the privacy of the variable. In such cases the variable can be used in that class's methods without qualification (prefixing with the class name), however this is not recommended and is considered poor style. If you wish to limit accessibility just to the defining class and its instances then the variable should be declared as `private`. 2 If you wish a class variable to be accessible to a subclass the variable needs to be declared as `protected` or `public`. In such cases the variable can be used in methods of any subclass without qualification (prefixing with the class name), however, this is not recommended and is considered poor style. 3 If you wish a class variable to be accessible to any unrelated class then the variable needs to be declared as `protected` or `public` and it must always be qualified by the class name within the methods of these unrelated classes. *Inadvertent effects* May be hidden in a subclass by a class variable of the same name declared in the subclass

Note that the lifetime and accessibility of variables should not be confused with lifetimes and accessibility of objects to which they may refer since there might be other references to those objects

14.5 Method and constructor arguments

These are not generally spoken of as variables, but conceptually they are variables, and their properties are very similar to those of local variables, as we will now summarise

Arguments are:

- declared in methods or constructors
- accessible only to the relevant method or constructor;
- created each time a method or constructor is executed
- destroyed when execution of the method or constructor ceases

5 General-purpose classes

The core of Java contains some important **general-purpose classes** which provide publicly available class methods and constants for a range of common tasks. Some of these classes are entirely static, and are written so that they cannot have subclasses, and no objects of the class can ever exist. All the work is done by class methods. You have seen some of these already in the course, for example, the `System` class contains a static variable `out`, which refers to the standard output stream (in the OUWorkspace, this is the Display Pane). The most common way to print out text from a Java program is to include a statement such as

```
System.out.println("Frogs of the world unite!");
```

This sends a message to the object `System.out` telling it to print a line containing the string that appears as the method argument.

At this stage you do not have to worry about the details of how to ensure that a class remains wholly static, but if you are interested, here is how it's done.

To stop a class having subclasses, we use the modifier `final`.

To stop a class having instances, we give it a `private` constructor, and then make a point of not invoking the constructor from within any methods in the class itself. The constructor cannot be invoked from other classes because they do not have access to it.

`System` is a good example of a general-purpose class that cannot have any instances. However, it contains various useful static facilities such as `System.out` that you can use from any Java program. This global accessibility is possible because `System` is defined in `java.lang`, a package that is always accessible to every Java program. Such facilities are all very well, but how are you supposed to know what facilities are provided by such classes? In *Unit 4* you explored the documentation for the OU Class Library which contains classes such as `OUDialog` and `OUColour`. In the next activity you are going to explore the documentation for the Java Class Libraries.

ACTIVITY 17

From the BlueJ Help menu, select Java Class Libraries. Selecting this option should open your web browser to offer documentation on the **Java Class Libraries**.

Use the browser, as described in the *Software Guide* and *Unit 4*, and open the documentation for the package `java.lang`.

1 From the left-hand frame of the browser window select the `System` class. Once you have done that you will see in the main frame of the window a description of its class methods and class variables. Find the entry for the class variable `out` (it will be under the heading Field Summary). What type is this class variable declared to be?

2 Now take a look at the documentation for the `Math` class, which is also in the `java.lang` package. You will notice that under the Fields heading, constants such as `PI` are declared (by the way, do not worry if mathematics is not your strong point: this example simply illustrates how class variables can be used to store useful constants, and π is just an example of a constant widely used in mathematics).

Classes such as `System` and `Math` are always automatically available in any Java program, since they are defined in `java.lang`, the only package automatically accessible to every Java program (including the OUWorkspace).

Execute the following statement in the OUWorkspace:

```
System.out.println(Math.PI);
```

Make sure that you type the class variable name `PI` in capitals, whereas the class name `Math` has only its initial letter capitalised). What value is returned?

3 Further exploration of the documentation of the `Math` class will show you that it also contains static methods for a wide variety of mathematical functions. For example, what does the class method `max()` do? Try executing

```
Math.max(42, 24);
```

in the OUWorkspace – what value is returned?

DISCUSSION OF ACTIVITY 17

1 The class variable `out` is declared as type `PrintStream`.

2 Execution of the following statement

```
System.out.println(Math.PI);
```

will output the value of the number π correct to 15 decimal places, like this.

```
3.141592653589793
```

3 The class method `max()` returns the largest number held by the two arguments. So the statement

```
Math.max(42, 24);
```

returns the number `42`

For more information about browsing Java Class Library documentation, see the M255 Software Guide

The above activity demonstrates how you can access a publicly accessible class variable (for example `Math.PI`), by giving the name of the class, followed by a dot, followed by the name of the variable. This class variable happens to refer to a primitive value (of type `double`). But of course, a class variable can just as well refer to a full-blown object. Indeed, there are several examples of class variables referring to generally useful objects which you have already been using. For example, **static constants** and methods are used to provide the colours and dialogue boxes that we have been using. Consider the `Frog` class. Frogs have colours, but all kinds of graphical objects need access to colours, and one simple way to make instances of colours readily available is to define commonly used colours as class variables in some easily accessible class, as you will see in the next activity where you revisit the documentation for the OU Class Library which you first looked at in *Unit 4*.

ACTIVITY 18

In this activity we ask you to consider the following statement:

```
OUColour myColour = OUColour.GREEN;
```

Execute the above statement in the OUWorkspace (you do not need to open a project).

Inspect `myColour`.

Is `GREEN` an instance variable, instance method, class variable or class method?

How has `GREEN` been declared in the `OUColour` class? To answer this question you need to look at the documentation for the `OUColour` class.

From the BlueJ Help menu select OU Class Library. Selecting this option should open your web browser to offer documentation on the OU Class Library.

From the list of classes in the left-hand frame, select `OUColour` and browse the documentation in the main part of the browser window to discover exactly what ... is

DISCUSSION OF ACTIVITY 18

If you open an inspector on `myColour` and then double-click on the green rectangle that is displayed in the inspector, you will see this (Figure 1)

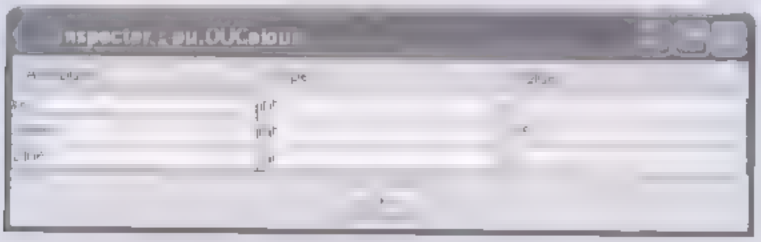


Figure 1 The components of `OUColour.GREEN`

This reveals how every colour in Java is represented as a mixture of three numbers representing the amount of red, green and blue in the colour. The minimum value is 0 (thus there is no red or blue in `OUColour.GREEN`) and the maximum value is 255 (you can see that `OUColour.GREEN` has as much green as a colour can get in Java).

Since the statement uses `GREEN` not `green()`, `GREEN` must be a variable, not a method and since `OUColour` is a class not an object, `GREEN` must be a class variable. The fact that `GREEN` is capitalised suggests that it is, furthermore, a constant.

The class documentation reveals that it is defined as follows

```
public static final OUColour GREEN
```

`GREEN` has been defined as a class variable, denoted by the use of the keyword `static`. Furthermore, `GREEN` has been defined as a constant, denoted by the keyword `final`. Because `GREEN` is a constant, evaluating the expression `OUColour.GREEN` will always return the identical colour object. However, the mere fact that a variable with a constant value was used to refer to an object will not *in itself* make that object immune from subsequent changes of state (for example, by sending it messages).

Fortunately, in the case of an instance of `OUColour`, there is no obvious message to send to it to alter its state. This is as things should be, since we would not want the class variable `OUColour.GREEN` to refer to an object that someone had somehow changed to look red, for example.

As an aside, one point may seem slightly confusing at first – we have a static variable (in this case a constant) `GREEN` that refers to an *instance* of the *same class*. This seems to go against the general rule that class methods and class variables should be usable without any instances of the class needing to exist. However, this general rule could be stated a little more accurately (if less memorably) if we said instead that class methods and class variables should be always usable without any explicit need to create any instances of the class. There is nothing to stop a class being programmed to create one or more instances of itself automatically when it is first used in a particular run of a program.

Colours are (relatively speaking) fun and easy to use. You can create an instance of your own colour, if you like, by mixing appropriate amounts of red, green and blue, as shown in the following example.

```
OUColour myColour = new OUColour(70, 100, 121);
```

You could use an instance of a colour like this to visibly change the colour of a frog, using `setColour()` as follows (if you had a project containing the `Frog` class open).

```
OUColour myColour = new OUColour(70, 100, 121);
// you may be able to improve on this colour!
Frog vanGogh = new Frog();
vanGogh.setColour(myColour);
```

The last two activities above focused on class *variables* (including class constants), rather than class *methods*. In the SAQ below you can review your knowledge of class methods. If necessary, use the OU Class Library documentation, as previously explained in Activity 18, to get more information about the relevant class.

SAQ 13

If you execute the statement `OUDialog.request("What is your favourite fish?");`

(a) What action results?

(b) What is the value returned?

(c) Is `OUDialog` a class or an object?

(d) Is `request()` an instance method or a class method?

(e) Where is `OUDialog.request()` accessible from?

ANSWER

(a) The effect of evaluating the statement

```
OUDialog.request("What is your favourite fish?")
```

is to display a dialogue box requesting input.

(b) The value returned is a string, consisting of whatever the user entered into the input box, or an empty string if the user didn't enter any characters. (As a step on the way, an instance of `OUDialog` is created, but this is not what is returned.)

(c) `OUDialog` is a class.

(d) Since `OUDialog` is a class, not an object, `request()` must be a class method, not an instance method.

(e) `OUDialog.request()` is accessible from anywhere in any program which imports the class `OUDialog`.

This use of a class method as a convenient way to create a new instance already tailored for some specific purpose (in the case of `OUDialog.request()`, to prompt the user in a certain way) is a common programming idiom. The general idea behind the idiom is that one or more class methods should be provided specifically to create new instances of that class. Typically, each such method returns an instance initialised in some particular way. (In the case of `OUDialog.request()`, although the idea is to create a dialogue box initialised in a particular way, a string is returned by the method invocation rather than the dialogue box itself.)

This kind of idiom can be much more flexible than using constructors explicitly, since any time you want a source of differently initialised instances you only need to create a

new class method. In the case of constructors, you can only have a single constructor of a given signature, whereas you can have as many class methods with the same argument types as you like, as long as you give the class methods different names

In the following SAQ, use the Java Class Library documentation (if necessary) to help you answer the questions

SAQ 14

If you execute the statement `System.out.println("Boo") ;`

(a) What action results?

(b) What is the value returned?

(c) Is `System` a class or an object?

(d) Is `out` a class, an instance variable, an instance method, a class method or a class variable?

(e) Is `println()` an instance method or a class method?

(f) Where is `System.out.println()` accessible from?

ANSWER...

(a) The effect of evaluating the statement `System.out.println("Boo")` is to display `Boo` in the Display Pane

(b) There is no message answer (the `println()` method is declared as `void`).

(c) `System` is a class It is a subclass of `Object` No instances or subclasses of it can be created but it has various useful class methods and class variables used for various system purposes such as input and output

(d) `System.out` is a class variable of the `System` class that references an instance of `PrintStream`. A `PrintStream` is an object used for managing text output – you do not need to know anything else about this class The static variable `out` is declared as follows

```
public static final PrintStream out
```

(e) `println()` is an instance method of `PrintStream`

`System.out` is a *class* variable and it references an *instance* of another class It does *not* reference a class! (Classes are not objects, and hence a variable cannot hold a reference that refers to a class) Thus, in

```
System.out.println("boo")
```

the message `println("boo")` is being sent to the *object* that the class variable `System.out` references For this reason, `println()` is an *instance* method.

(f) `System.out.println()` is accessible from any class in Java, since `System` is a class in `java.lang`, which is automatically accessible from any Java program.

6 Summary

After studying this unit you should understand the following ideas

- Work may be achieved in a Java program even if a message is not sent to an object (although the sending and receiving of messages is the most common way of carrying out operations)
- To achieve work in a program of any complexity objects will collaborate to interact both directly and indirectly
- If a method is overloaded or overridden, the JVM must choose one of several methods when it encounters the corresponding message at run-time The strategy for doing this will differ depending on whether the methods are overloaded or overridden
    - In the case of a message that corresponds to a number of overloaded methods it is the type of the message's arguments at compile-time that determines what method in the receiver's class hierarchy is invoked at run-time
    - In the case of a message that corresponds to an overridden method, it is the class of the receiver at run-time that determines what method is invoked at run-time
- There are primarily four ways of distributing code among methods to organise a sequence of actions
    - write the code in a single method,
    - distribute the code between two or more methods of the same class,
    - distribute the code between two or more methods of different classes
    - write a completely new class, whose methods would coordinate (orchestrate) a sequence of actions involving objects of some existing class(es) An instance of such a class is termed an *orchestrating instance*
- Class variables and class methods are declared using the access modifier `static`
- There is only a single copy of a class variable A class variable exists irrespective of whether any instances of the class have been created If declared as `private`, a class variable is only directly accessible by class methods, instance methods or constructors of the defining class
- Class methods are executed by invoking them, not on an object, but on the name of the class itself. They can be invoked irrespective of whether any instances of the class have been created
- Class methods declared as `private` can only be invoked by class methods instance methods or constructors of the defining class
- Class variables and class methods cannot be inherited
- A constant is a variable whose value is fixed and unchangeable. Constants are declared in Java with the `final` keyword

- Code refactoring – the rewriting of code to improve its design and maintainability, without changing its functionality – can be carried out according to three design principles, namely
  - eliminate duplicate code wherever possible;
  - ensure objects have responsibility for matters that concern them,
  - actions in a single method should be, wherever possible, at a single level of detail (that is focus in on activities or actions as tightly as possible)
- The Java Class Library contains some entirely static classes that cannot be subclassed. These classes contain general-purpose utility methods that can be used by all Java programs

LEARNING OUTCOMES

After studying this unit you should be able to

- describe the various ways in which work gets done in a Java program,
- use the design principles explained in this unit to select appropriate approaches to organising sequences of actions
- explain the difference between overloading and overriding a method
- explain the circumstances in which it would be appropriate to write a new class of object to orchestrate the coordination of a sequence of actions between objects of some existing class(es),
- explain the practical benefits of distributing responsibility for a complex sequence of actions between objects of relevant classes,
- begin to refactor code to achieve better design, remove code duplication and improve maintainability
- declare and initialise class variables using the keyword `static`;
- write and invoke class methods,
- use the keyword `final` to define constants,
- explain why classes such as `System` exist as purely static classes,
- describe the different types of variables available in Java and explain how and where they are used

Glossary

chained Constructors are said to be **chained** meaning that when an object is created the constructors of all its superclasses are always called, either explicitly or implicitly

class method A **class method** is a method declared with the keyword [illegible]. Class methods are associated with a class rather than with any of its instances In general a class method has no information about the existence of, or state of, any *instances* of its class Class methods are invoked directly on the name of the class, and not by sending a message. In Java classes (as opposed to their instances) are not objects, so class methods are *not* object-oriented and do not exhibit **inheritance** or **polymorphism**

class variable A **class variable** is a variable declared with the keyword [illegible] A class variable is associated with a *class* rather than with any of its *instances* A class only ever has one copy of each of its class variables In Java, classes (as opposed to their instances) are not objects hence class variables do not take part in **inheritance** See **qualified** for details of how class variables are accessed

constant A **constant** is a variable whose value is fixed and unchangeable. Normally the keyword `final` is used to make a value into a constant In addition, where only a single value is needed for a class, constants are typically declared as `static` However `static` is not always used for constants, as sometimes a situation needs to be modelled where each instance of a class has its own *different* constant value such as a serial number

constant instance variable Constants are usually declared as `final static` variables However, sometimes it makes more sense to define a constant as a `final` **instance variable** See **constant** for more information

design principle In contrast to a guideline or suggestion, the word *principle* is reserved for recommendations that apply universally or nearly universally In this course the term **design principle** is applied to principles governing how code should be organised. The authors reserve the right sometimes to violate design principles for pedagogical reasons!

direct interaction **Direct interaction** is not a formal technical term but a descriptive phrase used to describe a situation where one object has a reference to another, and this reference is used to affect the state or behaviour of the other object Contrast with **indirect interaction**

`final` The keyword `final` prevents a variable from ever having its value reassigned once it has an initial value

indirect interaction **Indirect interaction** is not a formal technical term, but a descriptive phrase used to describe a situation where one object affects the state or behaviour of the other object, but without actually having a reference to that object – i.e. some intermediary object is used. Contrast with **direct interaction**

member The term **member** is a convenient word sometimes used to cover all of the four following categories **class variables** **class methods** **instance variables** and **instance methods**

orchestrating instance An **orchestrating instance** is a separate object used to tie together the different parts of a complicated interaction that do not seem to belong to any single one of the objects involved.

overload A method name is said to be **overloaded** when a class has more than one method with the same name, but the methods differ in the *number of arguments*, or in the *order* or *type* of one or more arguments. Overloading should be contrasted with **overriding**.

overload-resolution When a method is **overloaded**, the compiler must decide which method signature the JVM should use to select a method at run-time. The process of picking the best match from a set of candidate methods is called **overload-resolution**.

override If a method has the same name and the same arguments (and return type) as an accessible method in a superclass, it is said to **override** that method. Contrast with **overloaded**.

qualified When accessing a **class variable**, it is generally necessary to access it via the name of the class using the dot notation (e.g. `MyClass.myStaticVariable`). This is described as using the **qualified name** of the class variable. This is not strictly necessary within **class methods**, **instance methods** or **constructors** of the class that defines the class variable, but it is good programming style and complements the use of `this` for accessing **instance variables**. Objects of other classes that might want to access an accessible class variable *must* qualify the variable with the class name.

refactoring **Refactoring** is when code is rewritten without changing its overall effect, but for the purpose of improving its design, removing code duplication, or improving maintainability. (See *Unit 6* for an alternative definition.)

visibility An object may contain an inherited instance variable, but be unable to access it directly because it was declared `private` in a superclass. However, a `public` accessor method declared in the superclass may allow the object to access this variable indirectly. In such a situation, the variable is said to lack **visibility**.

workspace variable A Java development environment that provides a workspace, like the one in M255, will have special **workspace variables**. Workspace variables behave like **local variables**, except that their lifetime lasts until you close or reset the workspace, rather than ending when a particular block has finished executing.

Index

C

chained 42

class methods 37

class variables 35, 48

constant 44

constant instance variable 45

D

design principle 17, 30

direct access to instance variables 47

direct interaction 9–10

F

`final` 44

G

general-purpose classes 51

H

helper methods 17

hiding 38

I

indirect interaction 10

initialisation of class variables 36

J

Java Class Libraries 51

`java.lang` 51

L

local variables 46

M

maintainability 24

masking 38

member 39

N

native methods 7

O

orchestrating instance 19

overload-resolution 14

overloaded 13, 15

override 15

P

`protected` 48

Q

qualified 37, 48

R

refactoring 25

S

sorts of variable 46

`static` 36

static constants 52

static variables 48

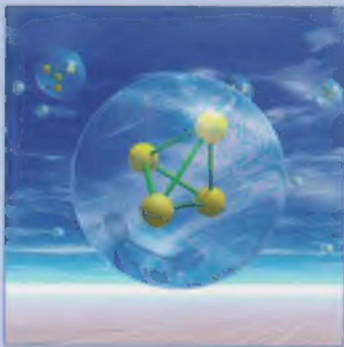
`System.out` 51

V

visibility 48

W

workspace variable 46



M255 Unit 7
UNDERGRADUATE COMPUTING

Object-oriented programming with Java

UNIT 7

Block 2

Unit 5 Dialogue boxes, selection and iteration

Unit 6 Subclassing and inheritance

▶ Unit 7 Code design and class members

Unit 8 Designing code, dealing with errors

M255 Unit 7
ISBN 978 0 7492 5499 5

9 780749 254995